Каспер Эрни

Программирование на языке АССЕМБЛЕРА для микроконтроллеров семейства i8051

Москва
Горячая линия - Телеком
2004

ББК 32.97
К28

Каспер Э.

К 28 Программирование на языке Ассемблера для микроконтроллеров семейства i8051. – М.: Горячая линия – Телеком, 2004. – 191 с.: ил.

ISBN 5-93517-104-X.

Изложены основы программирования на языке Ассемблера для популярного семейства микроконтроллеров i8051. Описаны особенности архитектуры микроконтроллеров семейства i8051. Приведены сведения о технологии разработки программ, системе и форматах команд. Книга содержит информацию о программировании некоторых типов задач, в том числе задач цифровой фильтрации сигналов, а также несколько рекомендаций о стиле программирования для начинающих программистов.

Для широкого круга специалистов, занимающихся разработкой промышленной и бытовой аппаратуры, радиолюбителей, может быть полезна студентам и аспирантам.

ББК 32.97

Адрес издательства в Интернет www.techbook.ru
e-mail: radios_hl@mtu-net.ru

Справочное издание

Каспер Эрни

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА ДЛЯ МИКРОКОНТРОЛЛЕРОВ СЕМЕЙСТВА I8051

Редактор Э. Н. Бадиков
Обложка художника Ю. С. Гусицкого
Верстка Э. Н. Бадикова, Е. З. Александрова

ЛР № 071825 от 16 марта 1999 г.
Подписано в печать 30.09.03. Формат 60×90/16. Гарнитура Times New Roman. Печать офсетная.
Уч.-изд. л. 12,75. Тираж 2000 экз. Изд. № 104

ISBN 5-93517-104-X

Предисловие

Автор, попав в начале 80-х гг. прошлого века по командировочным делам в Государственный оптический институт (ГОИ, бывший тогда головным научно-исследовательским учреждением Минооборонпрома), обмер, будто пораженный громом, красующимся на стене лозунгом с аршинными буквами "КАЖДОМУ ПРИБОРУ — СВОЙ МИКРОПРОЦЕССОР!". Этот призыв по сути дела был направлен в наступившее ныне будущее, когда массовый выпуск микроконтроллеров и наборов микросхем к ним освоен не только зарубежной, но и отечественной промышленностью.

Применение микроконтроллеров, начинавшееся в ту пору с цифровых часов и микрокалькуляторов, в настоящее время продолжает расширяться и оправдано не только малыми размерами, весом и энергопотреблением при высокой надежности и низкой стоимости этих микросхем. Главной причиной эффективности использования микроконтроллеров является их функциональная универсальность, которая обусловлена возможностью их программирования. Однако книги по программированию для микроконтроллеров на современном книжном рынке практически отсутствуют. Издания по микроконтроллерам в основном ориентированы на аппаратное обеспечение, хотя в них и приводятся сведения по архитектуре и системе команд, а иногда и по кросс-средствам для разработки и отладки программ.

Автор поставил перед собой достаточно скромную цель: поделиться практическим опытом программирования на языке Ассемблера для микроконтроллеров семейства i8051. Для автора работа по программированию для микроконтроллеров началась с заданного в лоб вопроса, сумеет ли он написать и отладить программу управления центрифугой. Самонадеянный ответ: “Не знаю, не пробовал. Наверно смогу!” не оставил путей к отступлению, и вот уже накоплен пятилетний опыт, оказавшийся успешным в производственном смысле.

Конечно, было бы хорошо издать фундаментальный труд, являющийся аналогом поваренной книги и состоящий из текстов функционально законченных блоков небольшого или среднего размера, которые могут оказаться полезными при разработке программ для самых разных изделий. Разработчикам и программистам нужны также книги о номенклатуре и характеристиках разнообразнейших микросхем, предназначенных для работы с микроконтроллерами. Из-за недостатка такого рода информации разработчикам микроконтроллерных устройств зачастую приходится "изобретать велосипед". Программирование драйверов для работы с микросхемами, подключаемыми к микроконтроллерам, также является интересной и важной задачей, не нашедшей отражения в книгах. Однако для ее решения пришлось бы собрать и обобщить разнообразный материал по очень большому количеству прикладных программ, что трудно сделать без риска быть обвиненным в разглашении фирменных секретов или, как говорят, "ноу-хау". Поэтому автор решил ограничиться учебным пособием.

Автор считает приятным долгом выразить признательность инженеру Волченкову А.В., сотрудничество с которым позволило разработать не только вышеупомянутую программу управления центрифугой, но и множество других не менее интересных и полезных программ. Его постоянная готовность помогать коллегам и устранять какие бы то ни было погрешности в своей части работы сделали совместную с ним деятельность автора вдохновляющей и плодотворной.

К.Э.

ВВЕДЕНИЕ

Программирование неразрывно связано с компьютерами, хотя за последние десятилетия проделана большая работа, чтобы эта связь стала более опосредованной. Стремление к переносимости и удобству пользовательского интерфейса возносит прикладные программы во все более высокие слои математического обеспечения, отделяемые от аппаратно-зависимых программ растущим количеством оболочек. Такое усложнение математического обеспечения в свою очередь требует все больших аппаратных ресурсов. Прогресс технических характеристик микропроцессоров и связанных с ними наборов микросхем вызывает с одной стороны восхищение, а с другой — тревогу. Моральное старение персональных компьютеров значительно опережает их физический износ. Хорошо еще, что благодаря микроминиатюризации этот прогресс не требует чрезмерных затрат материальных ресурсов, но вовлекаемые финансовые средства огромны. Экспоненциальный рост не может продолжаться безгранично. Можно только гадать, каковы будут результаты столкновения растущих аппетитов потребителей с неизбежными физическими и технологическими ограничениями прогресса в разработке компьютеров.

По сравнению с персональными компьютерами микроконтроллеры не так известны, потому что скромно прячутся внутри самых разнообразных устройств, в том числе и в компьютерах, в которых без микроконтроллеров не могло бы работать ни одно из периферийных устройств, таких как клавиатура, мышь, дисковод, принтер, сканер и т.д. Да и программирование для микроконтроллеров не является престижным по сравнению с программированием для персональных компьютеров. Но эта работа не менее важна и интересна, хотя и не столь заметна широкой публике. Особенности программирования для микроконтроллеров, с одной стороны, связаны с ограничениями вычислительных ресурсов, поскольку аппаратура должна стоить как можно дешевле. С другой стороны, программист должен

достаточно хорошо знать физику работы изделия, в котором используется микроконтроллер, поскольку разрабатываемые программы должны работать в реальном мире. В силу этих причин программы для микроконтроллеров являются штучным товаром, то есть непереносимы с одного типа изделия на другое, хотя тираж этих изделий может быть огромным. Ведь количество выпускаемых микроконтроллеров в десятки раз превышает производство микропроцессоров.

Поскольку программы для микроконтроллеров непосредственно связаны с их архитектурой, для обучения программированию необходимо выбирать наиболее широко распространенную систему команд. Поэтому автор выбрал разработанное корпорацией Intel семейство 8-разрядных микроконтроллеров i8051, практически ставшее к настоящему времени промышленным стандартом. Зарубежные и отечественные фирмы выпускают разнообразнейшие микроконтроллеры, совместимые с родоначальниками этого семейства. В пользу такого выбора говорит и то, что на рынке микросхем имеется богатый набор периферийных устройств, обеспечивающих взаимодействие микроконтроллеров этого семейства с разнообразной цифровой и аналоговой аппаратурой. К счастью, специфика архитектуры различных семейств микроконтроллеров не столь существенна по сравнению со спецификой языка Ассемблера, предлагаемого для изучения читателям этой книги. Так что освоившим программирование на Ассемблере для семейства i8051 будет не так уж трудно перейти на ассемблерное программирование для других семейств микроконтроллеров.

Книга доступна всем, кто теоретически знаком с программированием в объеме курса информатики средней школы и отладил хотя бы одну программу на любом из языков программирования. В связи с ограниченными ресурсами микроконтроллеров вся работа по подготовке программ для них проводится на персональных компьютерах. Разработка исходного текста, трансляция, компоновка и запись машинного кода в запоминающее устройство микроконтроллера производятся при помощи программ, работающих в операционной системе MS DOS. Поэтому желательно, чтобы читатель имел хотя бы минимальные навыки работы в этой системе.

Книга по программированию для микроконтроллера не может обойтись без сведений о его архитектуре. Помещенный в первой главе материал

ужат настолько, чтобы посвященные в начала компьютерных таинств читатели смогли освоить специфику программирования для микроконтроллеров i8051. Сведения о системе команд и о форматах команд и данных изложены более подробно. При этом команды представлены в соответствии с правилами их записи на Ассемблере. Сведения о машинных кодах команд даны в приложении.

Во второй главе излагаются общие сведения о технологии разработки программ на Ассемблере и приведен перечень директив, обеспечивающих управление процессами трансляции и компоновки. В связи с наличием большого количества синонимов некоторых директив в качестве основных были выбраны те, которые широко используются в диалектах Ассемблера для других микроконтроллеров или микропроцессоров.

Третья глава посвящена описанию одного из пакетов инструментальных программ, обеспечивающих разработку программ для микроконтроллеров семейства i8051 на IBM совместимых компьютерах. Этот не совсем современный, но надежный набор программ все еще широко используется программистами. Кроме способов работы с транслятором, библиотекарем и компоновщиком приведены сообщения об ошибках с переводом на русский язык.

В последующих четырех главах автор приводит сведения о программировании некоторых типов задач, вызывающих затруднения не только у начинающих программистов. В четвертой главе рассказывается о программировании таких арифметических действий, которые не могут быть выполнены одной командой. В пятой главе изложены методы программирования более сложных вычислений. Шестая глава посвящена программированию простейших способов фильтрации измерений. Седьмая глава касается программирования некоторых задач для устройств, работающих в реальном масштабе времени. Приведенные в этих главах примеры являются иллюстрациями особенностей программирования для микроконтроллеров. В большинстве случаев эти примеры заимствованы из реально работающих программ, хотя их работоспособность не гарантируется. Автор проверил их тексты только на отсутствие синтаксических ошибок.

В последней главе автор осмелился дать несколько рекомендаций о стиле программирования. Возможно некоторые из них заинтересуют

тех, кто уже имеет опыт программирования на Ассемблере, но они особенно важны для начинающих.

Наконец, следует упомянуть об использовании шрифтов в книге. В отличие от описательной части текста книги все примеры программ, синтаксические правила для команд и директив Ассемблера и общения с инструментальными программами записаны шрифтом, имеющим фиксированный шаг. В синтаксических описаниях для записи неизменяемых частей команд и директив используются прописные буквы прямого шрифта. Для записи изменяемых частей команд и директив используются строчные буквы курсива. В описаниях команд для вызова инструментальных программ необязательная часть выделяется квадратными скобками. Примеры информационных сообщений и сообщений об ошибках, выдаваемых этими программами на дисплей, выделены подчеркиванием.

Глава 1

Что нужно знать программисту о микроконтроллерах семейства i8051

1.1. Общие сведения об архитектуре i8051

Под архитектурой семейства процессоров подразумевается совокупность внутренних и внешних программно доступных ресурсов, система команд, система прерываний, функции ввода/вывода и протоколы обмена по магистралям. Начнем описание основателя семейства, микроконтроллера типа 8051, с перечисления 40 выводов интегральной схемы и кратких сведений об их назначении. Названия выводов выделены жирным шрифтом, а номера выводов приведены в скобках. Все эти сведения приведены только для общего представления, а программно доступные ресурсы будут обсуждаться далее по ходу изложения.

Источник питания подключается к выводам **Vss** (20) «земля» и **Vcc** (40) +5В. Для управления работой всех устройств в микроконтроллере используется генератор импульсов, который может работать от внешнего источника или автономно (в режиме самовозбуждения). В последнем случае к выводам **XTAL2** (18) и **XTAL1** (19) должен быть подключен кварцевый резонатор на частоту не более 12 МГц. После включения питания необходимо установить внутренние устройства микроконтроллера в исходное состояние подачей импульса на вывод **RST** (9). После этого микроконтроллер начинает работу с исполнения команды, записанной по нулевому адресу.

Для связи с внешним миром у микроконтроллера есть двунаправленные порты: 4 параллельных и один последовательный (дуплексный порт для приема и передачи). Программируя обращения по одному и тому же адресу порта, следует иметь в виду, что для чтения и записи используются разные устройства. Параллельные порты работают в байтовом формате, то есть имеют по 8 выводов: **P0** (32-39), **P1** (1-8), **P2** (21-28) и **P3** (10-17). Порты P0 и P2 используются также для выдачи адреса при обращении к внешним запоминающим устройствам. Выводы, к которым присоединены шины порта 3, могут использоваться:

- для последовательного порта в качестве принимающей **RxD** (10) и передающей **TxD** (11) линий,
- для ввода сигналов внешних прерываний **INT0#** (12) и **INT1#** (13),
- для счета внешних импульсов **T0** (14) и **T1** (15)
- для выдачи сигналов записи **WR#** (16) и чтения **RD#** (17) на внешние запоминающие устройства.

Кроме того для работы с внешними ЗУ используются выводы **PSEN#** (29), **ALE#** (30) и **EA#** (31).

Перейдем к внутреннему содержимому микроконтроллера типа 8051. При описании внутренних ресурсов символические обозначения приводятся, как это принято в Ассемблером и выделены жирным шрифтом, а адреса указаны в шестнадцатеричном коде.

Рассмотрим сначала некоторые особенности запоминающих устройств (ЗУ). История современных компьютеров началась с того, что фон Нейман предложил записывать программы в оперативную память. В отличие от этого для микроконтроллеров программа и данные размещаются в разных ЗУ (гарвардская школа). Программа и константы записываются в ПЗУ (постоянное ЗУ, или постоянную память) или по-английски ROM (Read Only Memory). А для хранения изменяемых данных используется ОЗУ (оперативное ЗУ, или оперативная память) или по-английски RAM (Random Access Memory). Для общения с внешними ЗУ в микроконтроллере имеются два 16-разрядных регистра: **PC** (Program Counter) и **DPTR** (Data PoinTeR). Первый из них (программный счетчик) используется только для чтения команд из ПЗУ, а второй (указатель данных) — для чтения данных из ПЗУ и ОЗУ и для записи в ОЗУ. Таким образом микроконтроллер может использовать адресное пространство до 65536 байт. Преимущество ПЗУ состоит в том, что его содержимое при выключении микроконтроллера не теряется. Некоторые микроконтроллеры семейства имеют внутреннее ПЗУ (в зависимости от модели его емкость может быть

от 2 до 32 Кбайт), что весьма удобно для разработчиков электронных схем. Есть модели микроконтроллеров, содержащие ЭППЗУ (постоянная память с электрической перезаписью), но запись и чтение производятся не одной командой, а специальной подпрограммой. Для хранения программ могут применяться ПЗУ различных типов. При массовом производстве используются ПЗУ с однократной записью (с "пережиганием"). Повторная запись возможна для ПЗУ со стиранием при помощи ультрафиолетового света или для ПЗУ с электрической перезаписью. Встроенные в микроконтроллер ПЗУ последнего типа удобны еще тем, что в случае необходимости можно запретить чтение записанной в них программы.

Внутреннее ОЗУ микроконтроллера имеет емкость всего 128 байт. Но адресное пространство у него 8-разрядное, то есть 256 байт. Старшие адреса предназначены для обращения к функциональным регистрам микроконтроллера. Впрочем, у наиболее совершенных моделей этого семейства емкость внутреннего ОЗУ равна 256 байт, но старшие адреса доступны только при косвенной адресации. Начальные ячейки оперативной памяти (32 байт) используются под однобайтовые регистры общего назначения: **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6** и **R7**. Их физические адреса зависят от содержимого 3 и 4 разрядов регистра **PSW** (Processor Status Word — слово состояния процессора, хотя его размер однобайтовый). Распределение адресов для этих групп ячеек, называемых банками 0, 1, 2 и 3, приведено в следующей таблице.

RS1	RS0	R0	R1	R2	R3	R4	R5	R6	R7	
0	0	00h	01h	02h	03h	04h	05h	06h	07h	банк 0
0	1	08h	09h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh	банк 1
1	0	10h	11h	12h	13h	14h	15h	16h	17h	банк 2
1	1	18h	19h	1Ah	1Bh	1Ch	1Dh	1Eh	1Fh	банк 3

Адреса от 20h до 7Fh программист может использовать по своему усмотрению. Часть адресного пространства от 80h до 0FFh занята для обращения к различным устройствам внутри микроконтроллера. Рассмотрим эти устройства, называемые также функциональными регистрами. Для каждого из них приведены мнемонические имена, а в скобках указаны их адреса.

Большинство функциональных регистров микроконтроллера 8-разрядные (в этом случае их разрядность не упоминается). Для выполнения арифметических и логических действий используется так называемый накопитель **A** или **ACC** (ACCumulator) (0E0h). Для операций умножения и деления используется дополнительный регистр **B** (0F0h). В зависимости от выполняемой команды в соответствии с результатом операции могут вырабатываться признаки, которые записываются в ранее упомянутый регистр **PSW** (0D0h). Адреса и назначение отдельных битов

этого регистра (каждый из таких битов называется **flag**) приведены в следующем перечне:

CY	(0D7h)	перенос из 7 (старшего) разряда
AC	(0D6h)	перенос из 3 разряда (середина байта)
F0	(0D5h)	флаг для использования программистом
RS1	(0D4h)	старший разряд номера банка
RS0	(0D3h)	младший разряд номера банка
OV	(0D2h)	переполнение результата
	(0D1h)	безымянный флаг (может использоваться программистом)
P	(0D0h)	признак нечетного количества единиц в коде результата

Из двух упомянутых ранее регистров программный счетчик никак не отображается на адресное пространство ОЗУ, дабы нечаянно не испортить его содержимое. Указатель данных состоит из двух байтов: старшего **DPH** (83h) и младшего **DPL** (82h) (H и L — от слов High и Low соответственно). Обращение к оперативной памяти может производиться в стековом режиме посредством регистра **SP** (Stack Pointer) (81h). Мнемонические имена регистров параллельных портов совпадают с обозначениями их выводов: **P0** (80h), **P1** (90h), **P2** (0A0h) и **P3** (0B0h). Последовательный порт производит передачу или прием через регистр **SBUF** (Serial BUFfer) (99h), а управление режимом его работы осуществляется при помощи регистра **SCON** (Serial CONtrol) (98h). В микроконтроллере имеется два счетчика с номерами 0 и 1, которые могут использоваться как для счета тактовых импульсов (режим таймера), так и для счета импульсов на внешних входах. Каждый из них состоит из двух байтов: старшего **TH0** (8Ch), **TH1** (8Dh) и младшего **TL0** (8Ah), **TL1** (8Bh). Для управления ими используется регистр управления **TCON** (Timer CONtrol) (88h). Чтобы обеспечить работу микроконтроллера с внешними объектами в реальном масштабе времени имеется система прерывания, для управления которой используются регистр разрешения прерываний **IE** (Interrupt Enable) (0A8h) и регистр приоритетов прерываний **IP** (Interrupt Priority) (0B8h). В микроконтроллере имеется возможность обращения не только в формате байта, но и к отдельным битам в функциональных регистрах и в некоторой части адресного пространства ОЗУ. Часть битов функциональных регистров имеет мнемонические имена. Но можно обращаться к битам и по имени регистра и номеру бита в байте, используя в качестве разделителя между именем и номером точку. Самый младший бит имеет номер 0, а самый старший — 7.

Приведем перечень начальных состояний всех функциональных регистров микроконтроллера на момент его инициализации:

ACC, B, PSW, DPH, DPL	00h
SP	07h

```
PO, P1, P2, P3                    FFh
SBUF                              xxxxxxxxb
SCON, TH0, TL0, TH1, TL1, TCON    00h
IE                                0x000000b
IP                                xx000000b
```

В перечне буквой х обозначено произвольное содержимое, а суффиксы b и h соответствуют шестнадцатеричному и двоичному кодам.

Следует заметить, что при программировании на Ассемблере информация об адресном пространстве ОЗУ нужна программисту только для оценки доступных ресурсов памяти. Попытки обращения к ОЗУ с несуществующими адресами (выше 7Fh) могут привести к непредсказуемым результатам. Впрочем при программировании на Ассемблере такие случайности исключены, хотя можно сделать это и преднамеренно.

Заканчивая общее описание микроконтроллера типа 8051, рассмотрим цикл исполнения команды. В зависимости от сложности выполняемых действий цикл исполнения команды занимает от одного до десяти тактов, длительность каждого из которых равна двенадцати периодам колебаний кварцевого резонатора. Если используется кварцевый резонатор на 12 МГц, то длительность такта равна 1 мкс. Цикл выполнения очередной команды начинается с чтения ее кода из ПЗУ по адресу, записанному в регистр PC (программный счетчик). После чтения каждого из байтов команды содержимое этого регистра увеличивается на 1, таким образом после чтения последнего байта команды программный счетчик содержит адрес первого байта следующей команды. Затем производятся чтение операндов, обработка полученной информации и запоминание результата операции. В зависимости от кода операции выполняемой команды те или иные действия могут не выполняться. В системе команд микроконтроллера имеются такие, которые могут изменить содержимое программного счетчика. Эти команды называются управляющими. Одни из них делают это в зависимости от результатов текущей или предыдущих операций, другие — независимо от выполнения каких-либо условий. За счет использования управляющих команд при выполнении программы микроконтроллер не только производит вычисления, но и производит логические действия типа выбора варианта или циклического повторения одних и тех же действий. Следует запомнить, что выполнение следующей команды начинается только после завершения текущей, то есть одновременное выполнение хотя бы двух команд невозможно. Однако во время выполнения программы отдельные устройства микроконтроллера могут работать автономно. Примеры программирования некоторых из этих устройств будут приведены позже.

1.2. Правила записи команд микроконтроллера семейства i8051 на Ассемблере

В стародавние времена для начала работы по программированию после описания архитектуры достаточно было привести подробную информацию о машинных кодах команд. Тогда программы записывались только на машинном языке. Такой способ разработки программ называется программированием в кодах (в жаргоне американских программистов по-прежнему используется термин “coding”). Запись восьмеричных или шестнадцатеричных кодов производилась на бланках, разбитых вертикальными линиями на колонки (графы). Каждая команда записывалась в одной строке. Первая графа была предназначена для адреса команды или числа. При записи команды во второй графе помещался код операции, а в следующих графах — адреса или значения операндов. Запись числа производилась в соответствии с его форматом. В бланке также отводилась графа для комментариев.

Ныне программирование в кодах вытеснено программированием на языках. Для программирования микроконтроллеров семейства i8051 можно использовать языки Ассемблер и усеченные Си и Паскаль. Интересно отметить, что даже проект новомодного языка Java начинался применительно к микроконтроллерам. Алгоритмические языки программирования имеют явные преимущества перед программированием в кодах благодаря применению символических имен. При выборе языка для программирования микроконтроллеров следует учесть ограничения на языки высокого уровня, накладываемые малым объемом ОЗУ, а в некоторых случаях и ПЗУ. В отличие от Ассемблера, для которого используется трансляция команд, записанных в символическом виде, языки высокого уровня используют компиляцию более сложных операторов, вместо которых подставляются стандартные заготовки из нескольких (иногда десятков) команд. Полученная таким образом программа далеко не оптимальна по быстродействию и требуемому объему памяти. Последующая за компиляцией оптимизация программы не позволяет достичь такой экономии ресурсов, какая получается при программировании на Ассемблере.

Алфавит Ассемблера состоит из прописных и строчных латинских букв, из цифр, знаков препинания и некоторых других символов, которые перечислены ниже в порядке возрастания их ASCII кода:

```
  ! # " $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

Кроме того, транслятор воспринимает символы, не имеющие отображения (пробел, табуляцию и пару символов – возврат каретки и перевод строки), но изменяющих положение следующих за ними символов текста. Транслятор производит грамматический разбор текста построчно. Каждая строка (аналог предложения) делится на лексемы (аналоги слов). В Ассемблере имеются лексемы 5 видов: специальные символы, имена (идентификаторы), числовые литералы, строковые литералы и комментарии. Специальные символы служат для обозначения операций в выражениях и в качестве разделителей других лексем.

Ассемблер не является языком свободной формы. Синтаксис Ассемблера в части записи операторов следует таким же правилам, какие применялись для программирования в кодах. Каждый оператор Ассемблера также занимает одну строку, состоящую из четырех полей

поле_метки поле_мнемокода поле_операндов ;поле комментариев

В качестве разделителей между полями можно использовать пробелы (не меньше одного) или знаки табуляции. В качестве разделителей между операндами (если их два или больше) используется запятая. Поле метки заполняется только в том случае, если в программе нужно сделать ссылку на данную строку. Поле мнемокода должно быть заполнено обязательно, так как в нем записывается производимое оператором действие. Необходимость записи в поле операндов зависит от мнемокода. Поле комментариев транслятору не нужно, поэтому остаток строки после точки с запятой не обрабатывается. Эта часть исходного текста существенна только для программиста. Поэтому в поле комментария можно использовать любые символы с кодом ASCII не меньше 20h, в том числе и русские буквы. Строка должна заканчиваться парой символов – возврата каретки и перевода строки.

В данной главе все команды микроконтроллера описаны в терминах Ассемблера. Полный перечень машинных кодов команд с соответствующими им символическими записями на Ассемблере приведен в Приложении 1. Во избежание опечаток текст приложения был получен как листинг трансляции программы, не имеющей алгоритмического смысла, за исключением того, что команды размещены в порядке увеличения кодов операций машинных команд.

В Приложении 2 приведен список команд в алфавитном порядке, которую автор рекомендует программистам использовать в качестве шпаргалки при повседневной работе. Для уменьшения объема этого списка команды сначала были сгруппированы по адресу приемника. При этом для обозначения выбора адреса источника было использовано стандартное синтаксическое обозначение с фигурными скобками и разделителем в виде вертикальной черты. Затем текст был сжат за счет использования стандартного синтаксического обозначения необязательного элемента, заключенного в квадратные скобки. В поле комментариев каждой команды приводится ее назначение, а за списком команд следуют пояснения условных обозначений. При дальнейшем чтении этой главы рекомендуется в качестве упражнения ознакомиться с обоими приложениями.

1.3. Форматы и способы адресации данных

Микроконтроллер типа 8051 работает с данными битового и байтового формата. При обработке данных можно обращаться не только к байтам во всех ЗУ, но и к отдельным битам функциональных регистров или части адресного пространства внутреннего ОЗУ. Есть несколько команд, работающих с данными двухбайтового формата. Часть команд обрабатывает каждый из битов байтовых операндов независимо от информации, содержащейся в других битах. Другая часть команд работает с байтами как с положительными целыми числами.

В командах обработки информации могут указываться один или два операнда. При явном использовании двух операндов один из них называется *источником* (source), а второй *приемником* (destination). Следует обратить внимание на то, что в поле операндов первым записывается обозначение приемника, а вторым — источника. При работе с одним операндом производится чтение операнда, его проверка или изменение и (при необходимости) запись. При работе с двумя операндами производится чтение источника и (при необходимости) приемника, вычисление и (при необходимости) запись результата по адресу приемника. Слова "при необходимости" означают, что необходимость действия определяется типом команды. После выполнения команды обработки информации содержимое слова состояния процессора может измениться в зависимости от результата вычислений. В описаниях команд, которые не осуществляют непосредственную запись в этот регистр, приводятся сведения о влиянии результата выполнения на содержимое отдельных битов PSW.

Адресация данных может быть *непосредственной* (immediate), *регистровой* (register), *прямой* (direct), *косвенной* (indirect), *индексной* (index) и *стековой* (stack). В первом случае данные содержатся непосредственно в команде, во втором — в регистре общего назначения, а в остальных — в одном из запоминающих устройств (внутренних или внешних). При адресации к памяти программ операнд может быть только источником, например непосредственная адресация допускается только для источника. В остальных случаях операнды могут быть как приемниками, так и источниками. Способ адресации операндов определяется кодом операции. Следует обратить внимание, что для некоторых операций одному и тому же мнемокоду соответствует несколько разновидностей машинных команд. Такое разнообразие связано с использованием разных способов адресации для одного и того же способа обработки информации и с использованием части байта кода операции для кодирования адресов операндов.

Для символического обозначения непосредственной адресации в источнике используются символ # и числовое выражение, по которому транслятор вычисляет и записывает в машинный код команды нужную константу. Допустимое десятичное значение константы от 0 до 255 для байтового формата и от 0 до 65535 для двухбайтового. В качестве непосредственного операнда можно записывать числа в десятичной, шестнадцатеричной, восьмеричной или двоичной системах счисления с суффиксами **d**, **h**, **o** (**q**) или **b** соответственно. По умолчанию транслятор считает число десятичным. Непосредственная адресация удобна тем, что не нужно тратить дополнительное время на чтение операнда, так как он хранится в команде. Увеличение размера машинной команды не столь важно, так как константу все равно нужно где-то хранить.

Для символического обозначения регистровой адресации используется имя любого из восьми регистров общего назначения. При регистровой адресации содержимое регистра обрабатывается как данные. Номер регистра общего назначения указывается в коде операции, вследствие чего для регистровой адресации не требуется увеличивать размер машинной команды. По этой причине регистровая адресация обоих операндов не допускается. Она чаще всего применяется для временного хранения промежуточных результатов вычислений и удобна тем, что на чтение из регистра и на запись в него не тратится дополнительное время.

Для символического обозначения прямой адресации используется имя операнда. В качестве такового можно записать имя какого-либо функционального регистра или имя данных, указанное в поле метки при резервировании адресного пространства ОЗУ. По этому имени транслятор

вычисляет фактическое значение адреса и подставляет его в машинный код команды. Таким образом, при использовании прямой адресации размер команды увеличивается. Поскольку при этом способе адресации к коду операции добавляются адресные байты, допускается прямая адресация обоих операндов. Следует иметь в виду, что в большинстве случаев прямая адресация к накопителю указывается только в ассемблерной команде, а в машинном коде нет его адреса. Он может появиться в машинном коде только тогда, когда его имя записано в обоих операндах. При использовании прямой адресации время выполнения команды увеличивается, а при адресации к портам – тем более.

Для обозначения косвенной адресации используется символ @, за которым следует выражение с именем регистра. При косвенной адресации содержимое регистра используется не в качестве данных, а в качестве адреса данных, то есть указателя на данные (pointer). Независимо от того, является ли операнд источником или приемником, содержимое регистра, используемого в качестве операнда, не изменяется. Для косвенной адресации могут использоваться только 2 регистра общего назначения (R0 и R1) или регистр указателя данных DPTR. Последний может использоваться во всем адресном пространстве внешних ЗУ, в то время как для регистров общего назначения максимальный адрес равен 255. При использовании косвениой адресации адреса регистров задаются кодом операции, так что увеличивать размер машинной команды не требуется. По этой причине косвенная адресация обоих операндов не допускается. Нельзя также использовать регистровую адресацию одного операнда и косвенную адресацию другого в одной и той же команде. Поскольку при косвенной адресации перед обращением к ЗУ необходимо сначала прочитать из регистра адрес операнда, то время выполнения команды увеличивается (особенно при обращении к внешним ЗУ).

Для индексной адресации используется сумма содержимого накопителя с содержимым программного счетчика или регистра указателя. Индексная адресация используется только для чтения из ПЗУ. Поскольку этот способ аналогичен косвенной адресации, то на Ассемблере он записывается как @A+ с последующим именем используемого регистра PC или DPTR. При выполнении команды с индексной адресацией необходимо вычисление адреса, поэтому время ее выполнения по сравнению с косвенной адресацией еще больше.

Стековая адресация также аналогична косвенной в том смысле, что адрес для обращения к ОЗУ берется из регистра, но в этом случае в качестве указателя используется содержимое регистра стека. Отличие состоит в том, что перед записью в ОЗУ содержимое указателя стека

автоматически увеличивается на 1, а после чтения из ОЗУ автоматически уменьшается на 1. Из-за автоматической коррекции содержимого указателя стека отпадает необходимость в записи одного из операндов команды. Но главное преимущество стековой адресации состоит в том, что в ОЗУ образуется очередь, которая обслуживается по принципу LIFO (Last In First Out— последним пришел, первым вышел). При использовании стекового способа адресации необходимо тщательно следить за балансом операций записи в стек и чтения из стека и правильно оценивать объем ОЗУ, используемый стеком. Надо иметь в виду, что помимо явного использования стековой адресации при обработке данных, некоторые управляющие команды неявно используют стековую адресацию.

Для уменьшения количества команд, приводимых в описаниях, использованы следующие обозначения. В командах с регистровой адресацией буквой ***n*** обозначена цифра, значение которой может быть в пределах от 0 до 7, а в командах с косвенной адресацией буквой ***i*** обозначена цифра, принимающая значение 0 или 1. Операнды с задаваемыми программистом именами в случае битового формата обозначаются ***flag***, а в случае байтового формата — ***src*** и ***dst***.

1.4. Форматы и способы адресации команд

Команды могут занимать от одного до трех байтов. Размер команды определяется кодом операции, записанным в первом байте, Дополнительные байты могут содержать адреса и/или данные. В символической записи команд как правило указываются все операнды. В машинных командах часть информации, необходимой для адресации, может содержаться в коде операции. При помощи байта кода операции можно закодировать 256 команд, но для микроконтроллера 8051 используется только 255. Код 0A5h зарезервирован для дальнейшего развития семейства.

Адресация команд (то есть их выполнение) по порядку их расположения в ПЗУ называется естественной. При завершении чтения очередной команды содержимое программного счетчика содержит адрес кода операции следующей команды. Команды, в результате выполнения которых может быть изменен естественный порядок исполнения команд, называются управляющими. Передача управления может происходить в зависимости от выполнения некоторых условий, тогда это называется *условной передачей управления* (conditional jump). Если команда всегда передает управление в другую часть программы, то это называется *безусловной передачей*

управления (unconditional jump). Некоторые из условных управляющих команд используют информацию, содержащуюся в слове состояния программы, а другие сами производят сравнение байтов или проверку содержимого битов ОЗУ. В операнде управляющей команды в общем случае должна содержаться информация для изменения кода в программном счетчике. Способы адресации управляющих команд в микроконтроллере типа 8051 различаются по дальности перехода на *короткие* (short), *абсолютные* (absolute) и *длинные* (long). Приведенные далее сведения по способам вычисления адреса, заносимого в программный счетчик управляющей командой, являются справочными. Для передачи управления программисту достаточно указать символический адрес перехода в соответствующем операнде ассемблерной команды.

При использовании короткого способа адресации в последнем байте команды содержится разность между адресом той команды, которой передается управления, и адресом команды, следующей за управляющей командой. Эта разность может составлять от –128 до +127. Для вычисления нового содержимого программного счетчика из содержимого последнего байта команды сначала формируется двухбайтовый код посредством записи старшего (знакового) бита во все разряды старшего байта. Затем двухбайтовый код прибавляется к содержимому программного счетчика. Такой способ часто называют *относительным* (relative).

Название абсолютного перехода унаследовано от предшествующей модели микроконтроллера, у которой объем ПЗУ был ограничен двумя килобайтами. При переходе к 64 Кбайт старое адресное пространство стали называть *страницей* (page). Поэтому 3 старших бита адреса перехода содержатся в коде операции, а 8 младших — во втором байте команды. Этот способ обеспечивает адресацию в пределах одной из 32 страниц ПЗУ, номер которой определяется 5 старшими разрядами кода операции. При абсолютном способе адресации 11 младших разрядов содержимого программного счетчика заменяются на содержимое адресной части команды. Для длинного перехода адресная часть команды состоит из двух байтов, содержимое которых заносится в программный счетчик. Короткий, абсолютный и длинный безусловные переходы обозначаются в мнемокодах команды начальными буквами S (Short), A (Absolute) и L (Long) соответственно.

Команды безусловного перехода можно разделить на команды без запоминания адреса возврата, команды с запоминанием адреса возврата и команды возврата. В последней разновидности команд безусловного перехода адресная часть отсутствует. Есть также команда безусловного

перехода, использующая индексную адресацию. Более подробно о способах адресации будет рассказано в описании управляющих команд.

Эту главу мы завершаем описанием команд, разделенных функционально на следующие 4 группы: команды пересылки информации, команды поразрядной обработки информации, арифметические команды и управляющие команды. Существует одна команда, которую нельзя отнести ни к одной из групп, так как она не делает ничего в течение одного такта:

```
NOP
```

Однако эта команда (No OPeration — нет операции) нужна для работы в реальном масштабе времени, чтобы обеспечить кратковременную задержку перед выполнением следующей команды.

1.5. Команды пересылки информации

Несмотря на пересылку данных в неизменном виде, эти команды осуществляют один из способов обработки информации. В качестве примера такой обработки можно привести сортировку. В командах пересылки используется все разнообразие способов адресации данных. Пересылка данных может осуществляться в форматах байта, половины байта, двух байтов и бита.

Начнем с байтового формата. Команда MOV (MOVe по-английски означает «передвинуть»; далее в аналогичных случаях мы не будем указывать специально, что дается перевод с английского) копирует содержимое источника в приемник (при выполнении этой команды первоначальное содержимое приемника теряется):

```
MOV    A, #src
MOV    A, Rn
MOV    A, @Ri
MOV    A, src
MOV    Rn, A
MOV    Rn, #src
MOV    Rn, src
MOV    @Ri, A
MOV    @Ri, #src
MOV    @Ri, src
MOV    dst, A
MOV    dst, #src
MOV    dst, Rn
MOV    dst, @Ri
MOV    dst, dst
```

Для засылки нуля в накопитель проще использовать команду очистки CLR (CleaR означает очистить):

```
CLR    A
```

Чтение и запись данных байтового формата при обращении к внешнему ОЗУ осуществляется при помощи команд MOVX, где буква X, по-видимому, означает eXternal (внешняя память):

```
MOVX  A, @Ri
MOVX  A, @DPTR
MOVX  @Ri, A
MOVX  @DPTR, A
```

Перед выполнением этой команды в соответствующий регистр нужно записать адрес.

Чтение данных из ПЗУ осуществляется при помощи команды MOVC, притом буква C скорее всего означает Code (программа).

```
MOVC  A, @A+DPTR
MOVC  A, @A+PC
```

Эти команды очень удобны для чтения из таблиц, записываемых в ПЗУ.

Запись в ОЗУ и чтение из него при помощи стекового способа адресации производятся командами

```
PUSH  src
POP   dst
```

Мнемокоды стековых команд соответствуют английским глаголам «затолкнуть» и «вытолкнуть».

Существует еще одна команда копирования XCH (eXCHange означает «обменять»), которая осуществляет обмен содержимого источника и приемника. В принципе обмен можно произвести при помощи трех команд пересылки. Следующие команды делают это за то же время, но занимают меньше места в ПЗУ и не требуют использования дополнительной ячейки ОЗУ:

```
XCH   A, Rn
XCH   A, @Ri
XCH   A, src
```

Есть также команда, обменивающая младшие половины байтов:

```
XCHD  A, @Ri
```

Здесь D означает Digit (четыре бита используются для двоичного представления десятичной цифры).

Одна из команд пересылки данных записывает два байта в регистр указателя данных:

```
MOV   DPTR, #src
```

Других команд для явной пересылки двухбайтовых данных нет.

Несколько команд пересылки информации работают в битовом формате. В команде MOV источником или приемником должен быть бит переноса C:

```
MOV   C, flag
MOV   flag, C
```

Для записи констант 0 и 1 используются команды очистки CLR и установки SETB (SET Bit означает «установить бит»):

```
CLR   C
CLR   flag
SETB  C
SETB  flag
```

Все команды пересылки не влияют на содержимое слова состояния программы, за исключением случаев пересылки информации в этот регистр или один из его битов.

1.6. Команды поразрядной обработки информации

Команды поразрядной обработки информации отличаются от команд арифметических операций тем, что работают с отдельными битами независимо от содержимого байта в целом. Такие команды есть для байтового и битового формата данных. Их также называют логическими командами, потому что с их помощью можно вычислять функции алгебры логики НЕ, И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ. Приведем для справки таблицу значений этих функций:

X	Y	/X	X.AND.Y	X.OR.Y	X.XOR.Y
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Все команды поразрядной обработки информации не влияют на содержимое слова состояния программы, за исключением случаев непосредственной записи результата в этот регистр или его биты.

Для получения обратного кода (логическая функция НЕ) при байтовом формате данных предназначена команда

```
CPL   A
```

Мнемоника этой команды ComPLement означает «дополнение», хотя в результате ее выполнения получается не дополнительный, а обратный код. Имеются две команды аналогичного назначения, работающие с отдельными битами:

```
CPL   C
CPL   flag
```

Кроме того, инверсия бита может быть использована в командах И и ИЛИ, как показано далее.

Команды для вычисления функции И в байтовом формате используют разнообразные способы адресации. По-видимому, мнемоника команды расшифровывается как ANd Logical.

```
ANL   A, #src
ANL   A, Rn
ANL   A, @Ri
ANL   A, src
ANL   dst, A
ANL   dst, #src
```

Эти команды могут использоваться, например, для очистки отдельных битов двоичного кода или для проверки наличия 1 в некотором наборе битов. Имеются две команды, вычисляющие функцию И в битовом формате:

```
ANL   C, flag
ANL   C, /flag
```

Символ косой черты во второй команде означает, что для вычисления логической функции используется инвертированное значение бита.

Аналогичный набор команд имеется и для функции ИЛИ в байтовом формате с мнемоникой OR Logical:

```
ORL   A, #src
ORL   A, Rn
ORL   A, @Ri
ORL   A, src
ORL   dst, A
ORL   dst, #src
```

Эти команды могут использоваться, например, для установки отдельных битов двоичного кода в 1. Имеются две аналогичные команды, вычисляющие функцию ИЛИ в битовом формате:

```
ORL   C, flag
ORL   C, /flag
```

Символ косой черты во второй команде означает, что для вычисления логической функции используется инвертированное значение бита.

Для функции ИСКЛЮЧАЮЩЕЕ ИЛИ аналогичные команды существуют только в байтовом формате. Наверно, соответствующая команда расшифровывается как eXclusive oR Logical:

```
XRL   A, #src
XRL   A, Rn
XRL   A, src
XRL   A, @Ri
XRL   dst, A
XRL   dst, #src
```

Эти команды могут использоваться, например, для изменения значения отдельных битов двоичного кода на обратное (toggle). Они могут также использоваться для проверки кодов на совпадение.

К командам поразрядной обработки информации можно отнести команды циклического сдвига влево и вправо, работающие с 8 битами (накопитель) или с 9 битами (накопитель + бит переноса):

```
RL    A
RR    A
RLC   A
RRC   A
```

Первая буква в мнемокодах этих команд означает Rotate (поворачивать), вторая указывает на направление (Left или Right), а третья — на участие бита переноса. При сдвиге влево во все биты накопителя кроме самого младшего записывается старое содержимое соседнего правого бита. При сдвиге вправо во все биты накопителя кроме самого старшего записывается старое содержимое соседнего левого бита. Если бит переноса не участвует в операции циклического сдвига, то при сдвиге влево в самый младший байт записывается старое содержимое самого старшего бита, а при сдвиге вправо в самый старший байт записывается старое содержимое самого младшего бита. При участии бита переноса его содержимое включается в цепочку циклического переноса, что позволяет осуществлять сдвиги содержимого многобайтовых кодов. К операции циклического сдвига на 4 разряда без участия бита переноса можно отнести команду

```
SWAP A
```

Она осуществляет обмен полубайтов содержимого накопителя, так что ее можно интерпретировать как сдвиг младшего полубайта на 4 разряда влево и сдвиг старшего полубайта на 4 разряда вправо.

Если перед выполнением команды сдвига очистить бит переноса, то сдвиг с участием этого бита может использоваться в качестве команды арифметической операции. Сдвиг вправо соответствует делению положительного числа на 2, притом в бит переноса записывается остаток от

деления. Сдвиг влево соответствует умножению положительного числа на 2, притом 1 в бите переноса сигнализирует о переполнении.

1.7. Команды арифметических операций

В ранее рассмотренных командах обработки информации кодирование содержимого отдельных битов байта не имело значения. В арифметических операциях нужно придерживаться строго определенных правил записи данных в соответствии с порядковыми номерами разрядов двоичного кода. В связи с ограниченными ресурсами микроконтроллера в нем используются только четыре арифметических операции с целыми числами. Отступления от правил кодирования приводят к неправильному выполнению арифметических операций.

В одном байте может быть закодировано 256 значений целого числа. При работе с положительными числами это соответствует значениям от 0 до 255. Все команды арифметических операций предназначены для работы с положительными целыми числами байтового формата, хотя команды сложения и вычитания в случае отсутствия переполнения обеспечивают получение корректного результата при специальном способе кодирования отрицательных чисел. Кодирование чисел более подробно рассмотрено в главе о программировании арифметических операций. При необходимости работы с числами, которые не могут быть представлены в байтовом формате, необходимо разрабатывать соответствующие подпрограммы. Выполнение операций умножения и деления с числами, имеющими произвольный знак, возможно при их представлении через знак и модуль и использовании соответствующих подпрограмм. Это могут быть собственные разработки или заимствования из хорошо проверенных библиотек других разработчиков.

Команда сложения работает с данными байтового формата, притом в качестве приемника всегда используется только накопитель:

```
ADD   A, #src
ADD   A, Rn
ADD   A, @Ri
ADD   A, src
```

Мнемоника этой команды соответствует слову ADDition (сложение).

Для работы с числами, которые не могут быть представлены одним байтом, используется команда сложения, учитывающая перенос, полученный при сложении предыдущей пары байтов:

```
ADDC   A, #src
ADDC   A, Rn
ADDC   A, @Ri
ADDC   A, src
```

Добавление буквы C к обозначению команды указывает на использование бита переноса (ADDition with Carrier).

Существует также команда сложения, при помощи которой производится увеличение заданного операнда на единицу (INCrement):

```
INC    A
INC    Rn
INC    @Ri
INC    src
```

Такая же команда есть и для работы с двумя байтами содержимого регистра указателя данных:

```
INC    DPTR
```

При помощи этой команды можно изменять содержимое указателя для чтения последовательности байтов из ПЗУ.

При сложении чисел, представленных двоично-десятичными кодами, после операции сложения нужно использовать команду десятичной коррекции суммы

```
DA    A
```

Двоично-десятичное кодирование имеет весьма ограниченное применение и потому далее не рассматривается.

Набор команд для вычитания гораздо уже. Команда вычисления разности существует только в варианте с вычитанием содержимого бита переноса (не хватило кодов команд!):

```
SUBB   A, #src
SUBB   A, Rn
SUBB   A, @Ri
SUBB   A, src
```

Мнемоника этой команды соответствует словам SUBtraction with Borrow (т. е. вычитание с учетом займа, так как при вычитании образуется заём, а не перенос). По этой причине перед вычислением разности младших байтов нужно обязательно очищать бит переноса, если нет уверенности в его содержимом. При вычислении разности старших байтов этого делать не нужно.

Существует также команда вычитания, при помощи которой производится уменьшение заданного операнда на единицу (DECrement):

```
DEC   A
DEC   Rn
DEC   src
DEC   @Ri
```

Команды уменьшения для работы с двухбайтовым форматом данных нет.

Результаты выполнения команд сложения и вычитания влияют на содержимое битов переноса, дополнительного переноса и переполнения в слове состояния программы. Результаты выполнения команд увеличения и уменьшения не влияют на содержимое слова состояния программы.

Команда умножения (MULtiplication) и деления (DIVision) работают при записи операндов в накопитель и регистр B. Для команды умножения порядок записи сомножителей в эти регистры не важен.

```
MUL   AB
```

Произведение имеет двухбайтовый формат. Младший байт произведения записывается в накопитель, а старший — в регистр B.

Для команды деления делимое должно быть записано в накопитель, а делитель — в регистр B:

```
DIV   AB
```

После выполнения команды в накопителе находится частное, а в регистре B — остаток. После выполнения команд умножения и деления в бит переноса заносится 0. Если старший байт произведения не равен нулю, то в байт переполнения заносится 1. При делении на 0 в байт переполнения также заносится 1.

1.8. Управляющие команды

Описание управляющих команд начнем с команд условного перехода. Эти команды используют только относительный способ адресации, поэтому для них будем использовать условное обозначение адреса перехода ***rel***. Для каждого условия существует пара команд, одна из которых осуществляет передачу управления при его соблюдении, а другая — при несоблюдении. В поле комментариев приводятся расшифровки мнемоники этих команд. Условием передачи управления может быть равенство или неравенство нулю содержимого регистра накопителя:

```
JZ    rel            ;Jump if Zero
JNZ   rel            ;Jump if No Zero
```

Можно также использовать в качестве условия перехода равенство бита переноса единице или нулю:

```
JC   rel            ;Jump if Carry
JNC  rel            ;Jump if Not Carry
```

Существуют команды, которые используют в качестве условия перехода равенство единице или нулю содержимого любого бита в функциональном регистре или адресуемого бита в ОЗУ:

```
JB   flag, rel      ;Jump if Bit
JNB  flag, rel      ;Jump if No Bit
```

Команда передачи управления по равенству бита единице имеет вариант с очисткой содержимого этого бита:

```
JBC  flag, rel      ;Jump if Bit and Clear
```

Команды с взаимоисключающими условиями позволяют обойти ограничения, связанные со способом адресации. Если адрес команды, на которую нужно передать управление, отличается от адреса следующей команды на большую величину (положительную или отрицательную), то можно использовать пару команд, первая из которых при соблюдении обратного условия передает управление через одну строку исходного текста, а вторая является командой безусловного перехода с абсолютной или дальней адресацией.

Перечисленные команды осуществляют переход в зависимости от результатов предыдущих вычислений. Однако есть управляющие команды, которые сами осуществляют вычисления для получения условий передачи управления. Мнемокод первой из таких команд – CJNE (Compare and Jump if Not Equal означает «сравнить и перейти, если не равно»). Это единственная команда микроконтроллера, имеющая 3 операнда. Ее четыре разновидности отличаются способами адресации источника и приемника:

```
CJNE  A,   src, rel
CJNE  A,   #src, rel
CJNE  Rn,  #src, rel
CJNE  @Ri, #src, rel
```

Команда вычисляет разность первого и второго операндов, но результат вычитания никуда не записывается, за исключением бита переноса. Передача управления по указанному адресу осуществляется при неравенстве операндов. При сравнении положительных чисел бит переноса устанавливается в 1, если первый операнд меньше второго. Если по адресу перехода записать команду передачи управления по содержимому бита переноса, то в результате одного сравнения можно выполнить три разные блока программы.

Другая команда — DJNZ (Decrement and Jump if Not Zero означает «уменьшить и перейти, если не равно нулю») уменьшает содержимое

первого операнда на единицу. Если операнд не равен 0, то управление передается по указанному адресу:

```
DJNZ  Rn, rel
DJNZ  dst, rel
```

Эта команда удобна для программирования цикла по счетчику. Перед началом цикла по адресу приемника надо записать число, равное количеству повторений цикла. Если во втором операнде записать адрес начала цикла и не изменять содержимое первого операнда другими командами в цикле, то заданный участок программы будет повторен заданное количество раз.

Перейдем к командам безусловного перехода JuMP без возврата. Они используют адресацию всех трех дальностей:

```
SJMP  rel
AJMP  adr11
LJMP  adr16
```

Цифры в условных обозначениях операндов последних двух команд указывают на количество разрядов адресной части их кодов.

Команда безусловного перехода с индексной адресацией позволяет изменять адрес перехода по содержимому накопителя:

```
JMP   @A+DPTR
```

При помощи этой команды можно осуществлять переходы по любому из 256 адресов относительно содержимого регистра указателя. Поскольку адрес передачи управления зависит от содержимого накопителя, этот вариант команды безусловной передачи управления, по сути дела, не является таковым. Эта команда может использоваться как переключатель, если в заданной области программы записать команды безусловного перехода на некоторые блоки программы. Так как эти команды будут занимать более одного байта, рассматриваемая команда может использоваться для передачи управления не более чем по 128 адресам.

Мнемоника команд безусловного перехода с возвратом CALL переводится как вызов. Эти команды не используют короткую адресацию:

```
ACALL adr11
LCALL adr16
```

Они применяются для вызова подпрограмм, после выполнения которых управление должно возвращаться команде, следующей за командой вызова. Для этой цели перед занесением адресной части команды вызова в программный счетчик его старое значение записывается в два байта ОЗУ, адресуемые указателем стека. Сначала в стек заносится младший байт

программного счетчика, а затем старший байт. Эта запись производится автоматически без использования каких-либо дополнительных команд.

Для корректного возврата к вызывающей программе последней выполняемой командой подпрограммы должна быть

```
RET
```

Мнемоника этой команды соответствует слову RETurn (вернуться). Команда не имеет адресной части, так как при ее выполнении в программный счетчик записываются два байта, адресуемые указателем стека (первым записывается старший байт, а вторым — младший). Программист должен обеспечить правильное содержимое указателя стека к моменту выхода из подпрограммы. Это требование означает, что при выполнении подпрограммы количество команд записи в стек должно быть равно количеству команд чтения из стека. Ввиду того, что адресное пространство стека размещается в ОЗУ, программист должен позаботиться о том, чтобы команды, использующие другие способы адресации, не производили запись информации в адресное пространство стека.

Другая команда, работающая точно таким же образом,

```
RETI
```

применяется для возврата к программе, выполнявшейся в момент аппаратного прерывания. Буква I в ее мнемокоде соответствует слову Interrupt (прерывание). Аппаратные прерывания передают управление фиксированным адресам с запоминанием адреса возврата. Программист должен разрабатывать программы обработки аппаратных прерываний по тем же правилам, что и подпрограммы.

Глава 2

Директивы ассемблера для микроконтроллеров семейства i8051

2.1. Общие понятия о процессах трансляции и компоновки

Приведенные в предыдущей главе команды могут быть переведены с языка Ассемблера в машинные коды микроконтроллера и далее в исполняемую программу при помощи двух программ, называемых транслятором и компоновщиком. Отложив до следующей главы вопрос о том, каким образом можно воспользоваться этими программами, рассмотрим возможности управления процессами трансляции и компоновки при помощи операторов исходного текста. При разработке программы на Ассемблере необходимо дать задание не только микроконтроллеру, но и транслятору с компоновщиком. Поэтому в Ассемблере существует два вида операторов: *команды*, которые превращаются транслятором в машинные коды, и *директивы*, в которых даются задания на трансляцию и компоновку. В отличие от команд директивы могут состоять из нескольких строк, то есть быть составными. Область действия таких директив заключена между открывающей и закрывающей строками. Часть директив также переводится транслятором в машинные коды или влияет на них, а некоторые директивы используются только для удобства работы программиста и потому без них в принципе можно обойтись. В отличие от ассемблерного программирования для компьютеров, когда нужно позаботиться еще и о взаимодействие программы с операционной

системой, для микроконтроллеров такой проблемы нет. Но это не облегчает, а усложняет работу программиста, так как он должен сам решать вопросы взаимодействия программы с внешней средой.

Написание и отладка программ содержит творческую и рутинную компоненты. Рутинная часть работы может быть автоматизирована настолько, насколько удается формализовать ее. Для того чтобы ассемблер и компоновщик могли выполнить формализуемую часть работы по переводу исходного текста программы в машинный код, программист должен включить в него директивы. Директивы являются средствами управления ассемблером и компоновщиком, каждый из которых, с одной стороны, достаточно "умен" (интеллекта в них ровно столько, сколько заложено разработчиками программного обеспечения), чтобы обходиться без мелочной опеки программиста, а с другой – вполне управляем. В Ассемблере директивы позволяют определять и использовать дополнительные программные объекты, не предусмотренные архитектурой процессора. Но они должны быть определены так, чтобы транслятор мог реализовать их имеющимися в микроконтроллере средствами.

Преобразование исходного текста в машинный код осуществляется в три этапа. Сначала этот текст должен быть записан в файл с расширением ASM при помощи обычного текстового редактора, не добавляющего никаких форматирующих символов. Затем текст нужно обработать транслятором, который в случае нарушения синтаксических правил выдает сообщения о местонахождении и типе каждой из ошибок. Если ошибок нет, то транслятор формирует объектный файл с расширением OBJ, а также, по желанию программиста, формирует файл протокола трансляции (листинг) с расширением LST. Объектный файл используется редактором связей (компоновщиком) для создания исполняемой программы, имеющей расширение TSK. Поскольку на каждом из этапов работы могут обнаружиться ошибки, приходится возвращаться назад для коррекции исходного текста. И только после получения машинного кода можно приступать к отладке программы.

Для того чтобы понимать сообщения транслятора и компоновщика об ошибках, нужно усвоить общие понятия о процессах трансляции и компоновки и о действиях, производимых транслятором и компоновщиком по директивам в исходном тексте. Знание принципов трансляции и компоновки существенно облегчает освоение разных диалектов Ассемблера, соответствующих особенностям как архитектуры процессоров, так и математического обеспечения разработки разных фирм.

Упомянутый выше листинг является текстовым файлом, предназначенным для программиста. В нем приводится не только исходный текст,

но и полученный машинный код в текстовом представлении. Для представления о задачах, решаемых при переходе от символического языка к машинному коду, рассмотрим листинг трансляции исходного текста на примере простой ассемблерной программы вычисления наибольшего общего делителя (НОД) двух положительных чисел по алгоритму Евклида. Он предписывает сначала разделить одно число на другое. Если остаток от деления равен нулю, то делитель является НОД. В противном случае нужно использовать в качестве делимого старый делитель, а в качестве делителя — остаток, и вернуться к операции деления.

Пусть два положительных числа (не превышающие 255) записаны в накопитель и регистр B. Тогда программа вычисления НОД может быть составлена всего из 3 команд. Эти команды работают в цикле, по выходе из которого искомое число будет находиться в регистре B. Лаконичность программы и отсутствие адресации к ОЗУ облегчают понимание механизма трансляции. Рассмотрим фрагмент листинга, представляющий результат трансляции программы:

```
1 0000  84      euclid: DIV  AB      ;деление
2 0001  C5 F0           XCH  A, B    ;перестановка
                                     ;частного и остатка
3 0003  70 FB           JNZ  euclid  ;возврат по
                                     ;ненулевому остатку
```

В самом листинге нет выделения ни жирным шрифтом, ни подчеркиванием, добавленных в этом примере автором. Жирный шрифт использован для показа исходного текста программы, который транслятор берет из файла с расширением ASM. А подчеркиванием выделены машинные коды команд, которые транслятор записывает в файл с расширением OBJ. В приведенном примере нет ни одной директивы. Но это не значит, что полноценную программу можно написать без директив. Отсутствие директив не приводит к тому, что транслятор и компоновщик работают “без руля и без ветрил”. В начале их работы устанавливаются режимы “по умолчанию”.

Как видите, результат трансляции программы представлен тремя группами столбцов. Первая колонка содержит десятичные номера строк исходного текста. Во второй колонке записаны шестнадцатеричные адреса команд в двухбайтовом формате. В последней группе колонок представлены байты машинных кодов команд в шестнадцатеричном виде. Примерно таким же образом записывались когда-то адреса и команды в стандартных бланках при программировании в кодах. Приращения адресов при переходе от строки к строке равны количеству байтов, отводимых для хранения соответствующих данных и команд. Убедиться

в соответствии кодов операций и способов адресации операндов мнемонической записи команд в исходном тексте можно при помощи перечня команд, приведенного в Приложении 1. Для первой и третьей команд транслятор выбрал коды операции однозначно по мнемокодам исходного текста, а для второй команды пришлось сделать выбор из 11 вариантов, имеющих один и тот же мнемокод, в соответствии со способом адресации второго операнда к функциональному регистру. Адресная часть второй команды выбрана по имени регистра B. Имя операнда третьей команды задано программистом, что несколько усложнило работу транслятора в части вычисления адресного бита команды. Это имя (euclid) было определено в поле метки первой команды, поэтому ему был поставлен в соответствие адрес 0. Для вычисления адресной части команды условного перехода транслятором использованы числовые значения адресов. После чтения управляющей команды содержимое программного счетчика будет равно 5, так что для возврата к началу программы нужно записать в ее адресной части –5, что в шестнадцатеричном коде байтового формата равно FBh.

Для обработки имен программных объектов транслятор ведет таблицу соответствия символических имен в исходном тексте программы их фактическим адресам. Эта таблица также выводится в листинг как текст:

```
Defined   Symbol Name   Value        References
Pre       BSECT         0000
Pre       CODE          0000
Pre       DATA          0000
  1       euclid        0000         3
Pre       RSECT         0000
```

В первой колонке таблицы приводится номер строки, в которой определено символическое имя, во второй — само имя, в третьей — соответствующий ему адрес и в четвертой — список номеров строк, в которых имеются ссылки на это имя. Имена упорядочены по алфавиту. В таблицу вошли четыре имени, которые отсутствуют в исходном тексте. Это стандартные имена секторов программы, поэтому в первой колонке таблицы для них вместо номера строки записано Pre (от predetermined — предопределено).

Если попытаться прочитать объектный файл при помощи текстового редактора в системе MS DOS, то из последовательности кодов 84 C5 F0 70 FB наверняка можно увидеть только букву “p” (латинскую строчную букву) вместо 70, а вид остальных символов будет зависеть от загруженной кодовой страницы. В частности, для кодовой страницы 866 (Россия) вместо 84 на дисплей выводится символ псевдографики (прямой угол с вершиной слева внизу), вместо C5 — буква “е”, вместо F0 — буква “П”

и вместо FB — буква "Ш". Кроме того, как будет показано в следующей главе, объектный файл содержит большое количество информации, необходимое компоновщику и совершенно не нужное программисту. Так что эти пять кодов нужно еще суметь найти в объектном файле. Из приведенного примера видно, насколько удобен листинг в части представления информации о результатах трансляции в текстовом формате.

Мнемокоды директив не совпадают с мнемокодами команд, но программисту желательно иметь бросающееся в глаза различие между ними. Поэтому для удобства чтения программ лучше начинать мнемокод директивы с точки (это допускается синтаксическими правилами транслятора семейства i8051). Многие директивы имеют синонимы, то есть директивы с разными мнемокодами обозначают одно и то же. В этом случае в качестве основного мнемокода приводится тот, который используется в других диалектах Ассемблера. Для того чтобы читатели могли изучать исходные тексты авторов, использующих разную запись директив, приводятся синонимы основного мнемокода.

А теперь приведем две простейшие директивы. Первая из них

```
.END
```

По этой директиве транслятор прекращает трансляцию. В следующей за этой строкой части исходного текста можно писать все что угодно, ибо транслятор не читает далее ни строчки. Обычно трансляторы разрабатываются для работы не с одним типом процессора, а со всем семейством. Для микроконтроллеров семейства i8051 выбор типа производится директивой

.CHIP *тип*

По умолчанию этой директивой производится ассемблирование для типа 8051.

2.2. Обработка имен транслятором и компоновщиком

Перевод мнемокодов и имен регистров ассемблерной команды в машинный код производится по заранее заданным таблицам с учетом способа адресации. Обработка символических имен, заданных программистом, представляет более сложную задачу. Программисту нужно понимать, каким образом эти имена обрабатываются транслятором и компоновщиком. Смысл обработки имен транслятором состоит в вычислении адресов, по которым программные объекты будут размещены в ОЗУ и ПЗУ микроконтроллера. По этим адресам определяются машинные коды команд, загружаемых в ПЗУ. Транслятор и компоновщик

(независимо от используемого языка программирования) решают задачу вычисления правильных адресов данных и команды в соответствии с их размещением в памяти.

Начнем ознакомление с проблемой вычисления адресов команд и данных с понятия о сегментировании памяти. В отличие от процессоров, для микроконтроллеров сегментирование связано не со способом вычисления исполнительного адреса, а с форматом обрабатываемой информации. Обращение к встроенному ОЗУ и к функциональным регистрам микроконтроллера может производиться как для чтения, так и для записи. Адресное пространство для данных в байтовом и битовом форматах от 00 до FF соответствует размеру байта адресации. В ОЗУ можно адресовать 128 байт (адреса от 00 до 7F) и столько же битов. Битовое пространство занимает байтовые адреса от 20 до 2F, то есть 16 байт того же самого ОЗУ. Остальные адреса используются для обращения к функциональным регистрам. Обращение к внешним ЗУ может производиться только в байтовом формате, притом к ЗУ команд — только для чтения. Адресное пространство внешних ЗУ ограничено двумя байтами адресации, что соответствует 65536 байт (адреса от 0000 до FFFF).

В Ассемблере стандартными сегментами (они называются также секциями) являются сегменты CODE (предназначен для программы) и DATA (предназначен для данных). Две других секции BSECT (битовая) и RSECT (регистровая) введены для программирования обращений к ОЗУ в битовом и байтовом форматах соответственно. Все эти названия имеются в таблице имен приведенного примера. Любая часть транслируемого текста должна относиться к какой-либо секции, то есть находиться между открывающей и закрывающей строками директивы определения секции:

```
.CODE
;команды и директивы секции кода
.ENDS
.DATA
;директивы секции данных
.ENDS
.RSECT
;директивы регистровой секции
.ENDS
.BSECT
;директивы битовой секции
.ENDS
```

Если в начале исходного текста не указано имя секции, то транслятор по умолчанию считает открытой секцию кода (именно поэтому нам удалась трансляция приведенного примера). Допускается вложение одних секций

в другие. Строка ENDS закрывает вложенную секцию. Если вложения секции не нужно, то для перехода к другой секции достаточно записать только открывающую строку директивы.

При желании можно задать дополнительные секции при помощи следующих директив:

```
имя: .SECTION   BIT   ;Описывается новая битовая секция
имя: .SECTION   REG   ;Описывается новая регистровая секция
```

Эти директивы позволяют программисту задавать имена секций. Для открытия секции, определенной программистом, нужно записать имя секции в поле мнемокода. Допускается использование не более 256 секций в одном файле.

Транслятор переводит данные и команды в машинный формат, располагая их в той последовательности, в какой они будут храниться в отведенном для них адресном пространстве секции. Поскольку программа состоит из нескольких секций, то вычисление адресов для записи двоичных кодов ведется для каждой секции отдельно. Начальный адрес в каждой из секций равен 0. Для использования счетчика текущего адреса в секции применяется предопределенное символическое имя $ (знак доллара) или * (звездочка). Числовое значение содержимого счетчика после трансляции строки исходного текста увеличивается на количество байтов, добавленных в адресное пространство текущего сегмента командой или директивой. Программист может использовать содержимое счетчика текущего адреса в выражениях, входящих в операнды команд или директив, и даже изменять его содержимое при помощи директивы. Запись заданного адреса в счетчик производится директивой

```
.ORG      адрес
```

Синонимом ORG является ORIGIN.

При обработке каждого оператора, которому соответствуют данные или команда, транслятор переводит их в двоичный код и записывает в отведенном для их секций адресном пространстве. Если оператор поименован (в поле метки есть имя), то транслятор записывает номер строки, имя объекта и содержимое счетчика адреса в таблицу имен. Конечно перед этим транслятор проверяет, нет ли этого имени в таблице. Если оно уже есть в таблице, то выдается сообщение об ошибке. Как только команда поименована, мы можем записывать в поле операнда управляющих команд ее адрес в символическом виде для передачи управления. Как только поименованы данные или отведенное для них место в памяти, мы можем записывать в поле операнда команды обработки информации их адрес в символическом виде (для чтения или записи).

Имя программного объекта может быть использовано в поле операнда только тогда, когда оно включено в таблицу символических имен. Если имя используется в той части программы, которая предшествует его определению, то транслятор не может использовать его при первом проходе, так что ему приходится читать исходный текст повторно после определения всех меток. Если и после повторного чтения программы не удалось обнаружить имя, записанное в операнде, то ввиду невозможности вычисления значения адресного выражения с этим именем транслятор выдает сообщение об ошибке.

Повторное использование имени в поле метки недопустимо, потому что нельзя создавать дополнительный программный объект с тем же самым именем. Исключения из правила уникальности используемых имен допускаются только в том случае, когда их повторное появление не нарушает этого правила. Это относится к директивам переопределяемых программных объектов и к локальным меткам. В таком случае при повторном использовании имени ранее созданный программный объект заменяется другим.

Имя должно начинаться с буквы и состоять из букв, цифр и может включать в себя некоторые другие символы. Его длина может быть очень большой, лишь бы оператор не вышел за пределы строки редактора, но только 32 первых символа будут иметь значение для таблицы имен. Большие и маленькие буквы в имени считаются различными. При соблюдении этих условий имя в поле метки считается нелокальным и включается в таблицу имен модуля. Имя, подставленное в поле метки, называется *меткой* (label). Метку можно начинать с любой колонки, если за именем следует двоеточие. Если двоеточие не используется, то первый символ имени должен располагаться в первой колонке.

Обычная метка наблюдаема (scope) во всем модуле. Однако при программировании можно пользоваться локальными метками, видимость которых ограничена двумя ближайшими нелокальными метками (предшествующей и последующей). Локальные метки могут быть использованы многократно, поэтому они не включаются в таблицу имен. Отличительным признаком локальной метки является символ $ в начале или конце имени, поэтому имя локальной метки может начинаться с цифры. Впрочем в качестве признака локальной метки можно использовать другой символ, который указывается директивой

.LLCHAR *символ*

Эта директива изменяет символ-определитель локальной метки.

До сих пор речь шла об именах, используемых в пределах одного программного модуля. Если программа состоит более чем из одного

модуля, то вычисление всех адресов на этапе трансляции невозможно. В связи с этим необходимо обратить внимание читателей на то, что адреса команд и данных на разных этапах подготовки программы к выполнению вычисляются с разной степенью определенности. Это касается размещения команд и данных в памяти и, как следствие, ссылок на адреса команд (из управляющих команд) или на адреса данных (из команд, обрабатывающих информацию). В связи с неполнотой информации о размещении данных и команд транслятор создает в объектных модулях три вида адресов:

- одна часть адресов определена окончательно и не подлежит изменению,
- другая часть адресов определена с точностью до размещения модулей в программе,
- третья часть адресов неизвестна, так как они находятся в других модулях (так называемые внешние ссылки).

Так что при трансляции каждого оператора, порождающего машинный код, транслятор прежде всего проверяет, какого рода адреса используются в операторе, по какому адресу этот код должен быть записан и какое символическое представление объекта программы должно быть сохранено для последующей сборки.

При использовании нескольких модулей большинство адресов, вычисленных транслятором, не являются окончательными и подлежит уточнению в процессе сборки. При сборке объектных модулей производится их перекомпоновка с целью объединения частей программы, содержащих одноименные секции, что позволяет вычислить все адреса. Имена секций сохраняются в объектных файлах для того, чтобы при сборке модулей объединять одноименные секции. При трансляции по умолчанию используется относительный режим, чтобы при сборке не происходило наложения адресных пространств. В относительном режиме сборка производится так, чтобы адресное пространство очередного модуля располагалось непосредственно за адресным пространством предыдущего. Таким образом для каждой из секций размер адресного пространства программы равен сумме адресных пространств модулей, если программист не задал смещения для собираемых частей секций.

При необходимости можно обеспечить фиксированные (не изменяемые компоновщиком) адреса в пределах 0 страницы ЗУ переключением транслятора в абсолютный режим директивой

```
.ABSOLUTE
```

Абсолютный режим допустим только для данных. Выполняемые команды всегда должны ассемблироваться в относительном режиме. Для возврата к относительному режиму используется директива

```
.RELATIVE
```

При переходе из одной секции в другую атрибуты ABSOLUTE и RELATIVE не изменяются.

В современных компьютерах большинство программ должно быть перемещаемыми, то есть работоспособными при их загрузке в произвольную часть ОЗУ. В микроконтроллерах все адреса окончательно определяются уже в процессе редактирования связей. Совершенно ясно, что транслятор может определить самостоятельно только первые два из трех перечисленных выше видов адреса. Притом адреса второго вида в объектном коде помечаются специальными символами, обнаружение которых используется компоновщиком для пересчета. Чтобы транслятор не выдавал сообщения об ошибках по третьему виду адресов, программист должен включить в программу соответствующие директивы. Связи между модулями осуществляются по данным и по управлению. Эти связи осуществляются при помощи символических имен, которые в модулях, вызывающих связь, объявляются как *внешние* при помощи директивы

`.EXTERNAL` *список*

В списке должны быть перечислены имена, определяемые в других модулях. Регистры могут быть заданы как внешние двумя путями. Если директива используется в регистровой секции, то внешние имена определяются как регистровые. Во всех остальных секциях необходимо перед каждым именем регистровой переменной в списке вставить слово REG. Синонимами EXTERNAL являются EXTERN и XREF.

Вызываемые имена должны быть определены в соответствующих модулях как *доступные* при помощи директивы

`.PUBLIC` *список*

Определив имя как доступное, можно ссылаться на него из других модулей. Синонимами PUBLIC являются XDEF и GLOBAL, хотя последний мнемокод в других диалектах Ассемблера используется для объявления и внешних, и доступных имен.

Существует еще один способ сделать имена доступными для использования другими модулями. Для этого используется директива

```
.GLOBALS ON
```

Транслятор считает все имена в поле меток команд следующих за этой директивой *глобальными*, доступными для ссылки из других модулей.

Эта директива не действует на локальные метки. Ее действие отменяется переходом к трансляции другого модуля или директивой

```
.GLOBAL OFF
```

Имена, использованные в разных модулях и не внесенные в соответствующие списки, не включаются в объектные файлы. В этом случае при компоновке редактор связей выдает сообщение об ошибке.

Использование промежуточного объектного файла при переводе исходного текста в исполняемую программу характерно для большинства современных языков программирования. Хотя получение исполняемой программы непосредственно из исходного текста в принципе возможно, по ряду причин сложилась практика двухступенчатого перехода. В случае программирования для микроконтроллеров это оправдано тем, что для разработки более или менее сложных программ удобно сначала отладить отдельные функционально законченные модули независимо друг от друга. Кроме того набор отлаженных объектных файлов, которые могут использоваться во многих программах, можно объединить при помощи программы-библиотекаря в специальный файл с расширением LIB. После отладки всех модулей их объектные файлы (включая библиотечные) собираются в один исполняемый файл для комплексной отладки программы.

Транслятор может обрабатывать несколько последовательно расположенных модулей исходного текста, составляющих один файл. Для этого в начале каждого модуля нужно записать директиву

```
.MODULE
```

а в конце модуля — директиву

```
.ENDMOD
```

Вложение модулей в другие модули не допускается. Транслятор осуществляет перевод модулей в машинный код независимо от других модулей. Количество модулей в файле не ограничивается. Выходной файл в этом случае имеет расширение PAK. Эта пара директив предназначена для получения объектных файлов, пригодных для работы с программой-библиотекарем. Их использование позволяет транслировать всю библиотеку в целом, вместо того чтобы транслировать каждую подпрограмму отдельно, а затем собирать объектные модули в один файл. Обычно библиотеки составляются из большого числа небольших по размерам программ. Когда компоновщик не может найти глобальное имя (идентификатор) в каком-либо из компонуемых файлов, он может продолжить его поиск в библиотеках.

2.3. Директивы резервирования памяти и инициализации данных

Директива резервирования памяти позволяет определить адрес информации для имени, указанного в поле метки этой директивы:

```
.DS    размер
```

Синонимами этой директивы являются DEFS и RMB. По директиве резервирования памяти значение счетчика текущего адреса в транслируемой секции увеличивается на число, указанное в операнде директивы, независимо от записи имени в поле метки. В битовой секции директива резервирует адреса для битов, а в остальных — для байтов. Инициализация данных при этом не производится. В резервируемую область не записываются никакие коды. Регистровые и битовые секции отличаются от секций кода и данных тем, что в них можно только отводить адресное пространство для данных, но нельзя инициализировать содержимое ОЗУ. Инициализация данных в битовой и регистровой секциях должна производиться программой сразу после включения микроконтроллера.

Инициализация данных может быть выполнена при помощи директив только для секций данных и кода. Директивы инициализации позволяют записать данные как в числовом, так и в символьном форматах. Если у операнда директивы инициализации чисел отсутствует суффикс, определяющий основание системы счисления, то принимается значение основания по умолчанию. Оно считается равным десяти, если до этого в тексте не была использована директива изменения системы счисления по умолчанию. Эта директива имеет вид

```
.RADIX    основание
```

в которой операнд может принимать следующие числовые или буквенные значения:

```
 2 или B (двоичная система),
 8 или O или Q (восьмеричная система),
10 или D (десятичная система),
16 или H (шестнадцатеричная система).
```

По умолчанию в этой директиве происходит возврат к десятичной системе. Следует отметить, что если задано основание системы счисления 16, то не существует способа описания десятичного или двоичного числа, поскольку суффиксы D и B являются допустимыми шестнадцатеричными цифрами.

Директива инициализации данных байтового формата имеет вид

```
.DB  значение
```

Синонимами этой директивы являются DEFB, BYTE, FCB и STRING. Операндом директивы может быть как одно значение, так и список значений, разделенных запятыми. Каждому элементу списка отводится один байт. При отсутствии операнда инициализируется одно нулевое значение. Допускается использование числового или символьного формата данных. Строки символов следует заключать в кавычки. Каждому символу строки отводится один байт (ограничители не включаются в объектный файл). Для включения кавычки в состав строки в качестве ограничителя следует использовать другой вид кавычек. При инициализации символов в машинном коде программы записываются числа, соответствующие загруженной в системе MS DOS кодовой странице.

Эта директива позволяет программисту записывать в любые места секции кода команды в машинных кодах. Таким образом, транслятор позволяет при желании вернуться к самому древнему методу программирования.

Для инициализации массива данных, имеющих одинаковые значения, можно использовать директиву

```
.BLKB   размер, значение
```

Количество инициализируемых байтов определяется размером. По умолчанию в каждом байте записывается нуль.

Существуют специальные директивы для инициализации данных символьного формата. Директива

```
.ASCII   строка
```

инициализирует коды символов ASCII, за исключением ограничителей строки. Если в строке появляется символ вертикальной черты (шестнадцатеричное 7C), то этот и последующие символы не инициализируются. Еще одна директива инициализации данных символьного формата

```
.DC      строка
```

использует в качестве ограничителей первый символ строки и следующий совпавший с ним. Инициализируются только символы, расположенные между ограничителями, притом в старший бит кода последнего инициализируемого символа записывается 1. Синонимом этой директивы является FCC.

Для ввода некоторых неотображаемых символов можно использовать их символическое представление при помощи пары букв:

```
"CR" или 'CR' – carriage return (возврат каретки)
"LF" или 'LF' – line feed (перевод строки)
```

```
"SP" или 'SP' – space (пробел)
"HT" или 'HT' – horizontal tab (горизонтальная табуляция)
"NL" или 'NL' – null (нуль)
```

Для того чтобы в операндах директив инициализации можно было использовать приведенные обозначения неотображаемых символов, используется директива

```
.TWOCHAR ON
```

По умолчанию используется режим выключения ввода неотображаемых символов. При необходимости можно выключить этот режим директивой

```
.TWOCHAR OFF
```

Чтобы отличать символ от числа, когда-то было принято записывать единицу в старший бит каждого кода символа при инициализации символьных данных директивами ASCII или DB. Для включения и выключения режима записи 1 в седьмой бит кода символа используются директивы

```
.BIT7 ON
.BIT7 OFF
```

По умолчанию режим записи 1 в седьмой бит кода символа выключен.

Существуют директивы инициализации данных, состоящих из двух байтов (слов). Инициализация отдельных значений или списков значений осуществляется только в числовом формате директивой

```
.DW  значение
```

Синонимами этой директивы являются DEFW, WORD и FDB. Для инициализации массива одинаковых значений в формате слова используется директива

```
.BLKW   размер, значение
```

Она инициализирует заданное количество 16-разрядных слов с заданным значением. По умолчанию инициализируются значения нуль.

Поскольку система команд I8051 не работает с данными в формате слова, то расположение старших и младших байтов может задаваться программистом произвольно. Для работавших с системой команд IBM PC привычно записывать старший байт по старшему адресу, притом адрес младшего байта должен быть четным. Приведенные директивы инициализации данных в формате слова располагают старший байт по младшему адресу без соблюдения четности адресов.

Существуют также директивы инициализации числовых данных в форматах LONG, FLOAT и DOUBLE, но для микроконтроллеров в подавляющем большинстве случаев использование таких форматов является экзотикой.

2.4. Использование выражений в операндах

Для удобства программиста в трансляторе имеется возможность вычисления числовых значений операндов, записываемых в виде выражений. Следует помнить, что эти выражения вычисляются только на этапе трансляции. В представленном ниже перечне для каждой вычислительной операции указано символическое обозначение, уровень приоритета и назначение. Ограничителями буквенных обозначений вычислительных операций являются точки. Операции, имеющие наивысший уровень приоритета, являются унарными (работают с одним аргументом).

Операция	Приоритет	Назначение
+	7	Подтверждение знака операнда
-	7	Изменение знака на противоположный
.NOT.	7	Поразрядная инверсия
>	7	Выделение старшего байта слова
<	7	Выделение младшего байта слова
**	6	Возведение в степень (положительные числа)
*	5	Умножение положительных чисел
/	5	Вычисление частного от деления положительных чисел
.MOD.	5	Вычисление остатка от деления положительных чисел
.SHR.	5	Сдвиг двоичного кода вправо
.SHL.	5	Сдвиг двоичного кода влево
+	4	Сложение
-	4	Вычитание
.AND.	3	Поразрядное логическое И
.OR.	2	Поразрядное логическое ИЛИ
.XOR.	2	Поразрядное логическое ИСКЛЮЧАЮЩЕЕ ИЛИ

При выполнении вычислительных операций наивысший приоритет имеют круглые скобки. Вычисления осуществляются при помощи 80-разрядных двоичных целых чисел, за исключением операций возведения в степень, в которых используется 8-разрядное значение показателя степени. Однако в качестве операндов используется целая часть выражения (за исключением форматов FLOAT и DOUBLE) в двоичном представлении с ограниченным количеством разрядов (8 — для байта и 16 — для слова). При возведении в степень первым операндом является основание, а вторым — показатель степени. При сдвиге первым операндом является

величина сдвига, а вторым — сдвигаемый код. Освобождаемые при сдвиге разряды кода заполняются нулями.

Уровень приоритета 1 (самый низкий) имеют операции сравнения числовых значений, результатом которых является логическое значение. Истинное значение представляется набором единиц во всех разрядах байта, а ложное — набором нулей. В представленном далее перечне приводятся операции сравнения периода ассемблирования. Приведенные в перечне высказывания являются условиями истинности выражения с соответствующей операцией сравнения.

Операция	Приоритет	Условие истинности
.EQ.	1	первое число равно второму
.GT.	1	первое число больше второго
.LT.	1	первое число меньше второго
.UGT.	1	первое число выше второго
.ULT.	1	первое число ниже второго

При сравнении чисел следует учитывать, что для заведомо положительных чисел и для чисел со знаком используется разное кодирование. Поэтому для сравнения чисел со знаком должны использоваться 2-я и 3-я операции, а для сравнения заведомо положительных чисел — 4-я и 5-я.

2.5. Директивы условной трансляции

Для управления трансляцией существуют директивы выбора транслируемого текста в зависимости от выполнения тех или иных условий. Эти директивы составные, то есть могут состоять из двух или трех строк. В первой строке оператора формулируется условие трансляции следующего за ним текста. Здесь он приведен в условном виде из-за большого количества разновидностей открывающего оператора директивы условной трансляции. Все эти разновидности будут описаны далее. Вторая строка оператора используется для обозначения начала текста, который транслируется при несоблюдении условия. Если альтернативного текста нет, то этот оператор не нужен. Последняя строка закрывает директиву условной трансляции.

```
.IF условие
;транслируется при соблюдении условия
.ELSE
;транслируется при несоблюдении условия
.ENDIF
```

Синонимом закрывающей строки условного оператора является ENDC. Допускается вложение директив условной трансляции (до 248 уровней). Транслятор проверяет соответствие пар открывающих и закрывающих строк. В случае отсутствия парности он выдает сообщение об ошибке. В составе открывающих строк директив условной трансляции имеется не только множество синонимов, но и большое количество антонимов (мнемокодов с противоположным условием).

Начнем с директив, проверяющих числовое значение выражения, записанного в поле операнда.

```
.IFNZ значение
```

Условием выполнения приведенного мнемокода является неравенство операнда 0. Синонимами этой директивы являются IF, IFN и COND. Антоним этой директивы проверяет равенство операнда 0.

```
.IFZ значение
```

синонимом которой является IFE.

Аналогичные директивы проверяют истинность логических выражений, сравнивающих числа.

```
.IFTRUE сравнение
```

Условием выполнения приведенного мнемокода является истинность логического выражения. Синонимом этой директивы является IFNFALSE. Данная директива фактически является эквивалентом IFNZ. Антонимом директивы IFTRUE является директива

```
.IFFALSE сравнение
```

Условием выполнения ее мнемокода является ложность выражения. Синоним этой директивы IFNTRUE. Данная директива фактически является эквивалентом IFZ.

Есть директивы с поверкой совпадения строк, заданных первым и вторым операндами.

```
.IFSAME строка1,строка2
```

Условием выполнения приведенного мнемокода является идентичность строк. Синонимом этой директивы является IFNDIFF. Если строка содержит пробелы, то она должна быть заключена в кавычки. Антоним этой директивы проверяет отсутствие идентичности строк.

```
.IFNSAME строка1,строка2
```

Синонимом этой директивы является IFDIFF.

Имеются директивы, проверяющие операнд на совпадение с каким-либо именем в таблице программных объектов, которую составляет

транслятор. Условием выполнения приведенного мнемокода является наличие операнда этой директивы в таблице имен.

```
.IFDEF  имя
```

Антонимом этой директивы является IFNDEF. Поскольку имена в таблице могут быть внешними и внутренними, существует директива проверки принадлежности операнда к внешним именам

```
.IFEXT  имя
```

Антонимом этой директивы является IFNEXT. Если в директивах проверки внешних меток условие не выполнено, то транслятор выдает сообщение об ошибке.

При наличии имени в таблице программных объектов может потребоваться проверить, является ли адрес этого имени абсолютным или относительным:

```
.IFREL  имя
```

Условием выполнения приведенного мнемокода является наличие перемещаемого адреса для заданного имени. Синонимом этой директивы является IFNABS. Внешние метки являются перемещаемыми, поэтому при использовании в качестве операнда внешнего имени условие считается выполненным. Проверка наличия абсолютного адреса для заданного имени осуществляется директивой

```
.IFABS  имя
```

Синонимом этой директивы является IFNREL. Если в этой директиве используется операнд, соответствующий внешнему имени, то условие считается невыполненным. По невыполнению условий двух этих директив транслятор также выдает сообщение об ошибке. Выдача ошибки при неправильном определении внешнего имени или связанного с ним адреса нужна потому, что компоновка объектного модуля в этом случае приведет к ошибкам в машинном коде.

Кроме перечисленных директив условной трансляции существуют директивы аналогичного типа, используемые в сложных текстовых подстановках.

2.6. Директивы подстановок

В Ассемблере могут использоваться подстановки, которые являются очень удобным средством программирования. При помощи директив подстановок программист создает поименованные объекты, которые могут быть использованы в тех или иных местах исходного текста.

Корректное использование подстановок существенно уменьшает влияние описок. Название языка Ассемблер (сборщик) связано не столько с возможностями трансляции (трансляция осуществлялась уже в ЯСК — языке символического кодирования), сколько с возможностью сборки программ из отлаженных текстовых заготовок.

Существуют два вида простых подстановок: числовая и текстовая. Директива числовой подстановки приписывает имени, указанному в поле метки, заданное операндом директивы числовое значение.

```
имя:  .VAR  значение
```

Синонимом этой директивы является DEFL. Числовая подстановка может переопределять имя, то есть этой директивой разрешается изменять числовое значение, приписанное имени.

Директива текстовой подстановки присваивает имени, указанному в поле метки, текстовое значение:

```
имя:  .EQU  текст
```

Синонимом этой директивы является EQUAL. Переопределять имя, заданное текстовой подстановкой, нельзя. Операнды в обеих директивах представляют собой символы исходного текста. Транслятор обрабатывает эти подстановки по-разному: для числовой подстановки трансляция операнда осуществляется до подстановки, а для текстовой — после подстановки.

Еще одна директива текстовой подстановки позволяет программисту назначить произвольное имя регистру или адресуемому биту. Это имя может иметь до десяти символов и запоминается в отдельном буфере так, что таблица символов не содержит его, и оно не может быть проверено. Таблица регистров заполняется в следующем порядке:

1.- Предопределенные имена регистров;

2.- Назначенные программистом имена регистров;

3.- Назначенные программистом имена битов.

Назначение имени производится директивой

```
имя:  .REG   аргумент
```

Синонимом этой директивы является REGISTER. В качестве аргумента директивы могут использоваться предопределенные регистры или биты или ранее определенные имена регистров. Для обозначения бита в аргументе указывается имя регистра общего назначения или функционального регистра, а затем номер бита, отделяемый от имени регистра точкой. Максимальное количество назначенных имен равно 512.

Гораздо более эффективными являются сложные текстовые подстановки. Обычно для такого рода конструкций языка используется термин

МАКРОС. Это слово, заимствованное из греческого языка, означает нечто большое. Макросом является последовательность строк исходного кода, которая будет подставляться вместо одной исходной строки. Прежде чем макрос сможет быть использован, его надо определить при помощи составной директивы сложной текстовой подстановки

```
имя:   .MACRO аргументы
;текст подстановки
       .ENDM
```

Синонимом закрывающей строки для сложной текстовой подстановки является MACEND. В списке аргументов не допускается использовать пробелы, разделителем аргументов является запятая. Отсутствие аргументов объявляется одиночной запятой. В текст подстановки могут быть включены директивы условного ассемблирования. В этом случае в конце блока, ассемблируемого по выполнению условия, нужно использовать директиву

```
.MACEXIT
```

Говорят, что эта директива вызывает непосредственный выход из макроса, хотя на самом деле прекращается подстановка текста с заданным именем. Все условно ассемблированные величины запоминаются тем же образом, что и при завершении макроса. В сложных текстовых подстановках допускается использование меток. Уникальность используемых меток может обеспечиваться двумя способами: явным и неявным. Для явного способа программист должен записать их в определении сложной текстовой подстановки таким образом, чтобы в результате подстановки они были уникальными. В этом случае в состав метки можно включать аргумент. Признаком неявного способа служит символ # в конце имени. В процессе трансляции ассемблер вместо знака # подставляет состоящий из 3 цифр номера подстановки. В этом случае в тексте подстановки длина метки вместе с символом номера не должна превышать 30.

Транслятор хранит определение сложной текстовой подстановки и вставляет ее вместо того оператора исходного текста, в мнемокоде которого записано имя сложной текстовой подстановки. Операнды замещаемого оператора подставляются в текст подстановки вместо аргументов, заданных в определении сложной текстовой подстановки. Указанные в макроопределении аргументы являются его формальными параметрами. При выполнении подстановки формальные параметры заменяются на фактические значения, содержащиеся в операндах вызова сложной текстовой подстановки. После выполнения подстановки производится трансляция подставленного текста в машинный код.

Фактические значения аргументов сложной текстовой подстановки могут иметь любой тип: прямые, косвенные, символьные строки или

регистры. Пробелы в аргументах недопустимы, за исключением строк в кодах ASCII; в этом случае строка должна быть заключена в кавычки. Если имена формальных аргументов идентичны, то аргументы будут передаваться любым вложенным макросам. Вложение макросов ограничивается только объемом доступного транслятору пространства памяти. Если в качестве фактического значения необходимо использовать запятую, то в качестве разделителя фактических параметров можно использовать скобки. Для этого необходимо перед определением сложной текстовой подстановки использовать директиву

```
.MACDELIM скобка
```

В качестве операнда этой директивы разрешается использовать символы открывающих скобок (обычной, фигурной или квадратной). При вызове подстановки фактические значения аргументов должны быть заключены в открывающую и закрывающую скобки того вида, который указан в директиве.

При подстановке может осуществляться замещение части текста, обозначенной как формальный параметр, на фактическое текстовое значение. Для этого аргумент должен отделяться от остального текста символом конкатенации строк, в качестве которого используется символ вертикальной черты (| — шестнадцатеричный код 7С). Выражения с операцией конкатенации (слияния строк) могут использоваться только внутри определения сложной текстовой подстановки. Для того чтобы отличить от числового значения текстовое обозначение числа, последнее записывается в угловых скобках <>, при этом пробел между символом конкатенации и открывающей угловой скобки недопустим.

Транслятор осуществляет проверку количества операндов в вызове количеству аргументов в определении сложной текстовой подстановке. Для включения и выключения этой проверки можно использовать директивы

```
.ARGCHK ON
.ARGCHK OFF
```

Чтобы использовать в вызове подстановки меньшее количество параметров, используется директива ARGCHK OFF. Первоначально проверка включена.

Для проверки наличия аргументов в операнде вызова сложной текстовой подстановки в ее определении можно использовать условную директиву

```
.IFMA  выражение
```

По этой директиве транслятор вычисляет значение выражения и ищет в операнде вызова подстановки аргумент с полученным порядковым

номером. При наличии аргумента для подстановки используется текст, предшествующий директиве ELSE, а при отсутствии — следующий за нею. Условие отсутствия аргументов может быть задано посредством условия <выражение>=0. В этом случае, если в строке вызова макроса никакие аргументы не присутствуют, то подставляется текст, предшествующий директиве ELSE, а при наличии — следующий за нею. Антонимом этой директивы является IFNMA.

При использовании рекурсивного вызова сложной текстовой подстановки (использование в тексте подстановки своего собственного имени) для завершения подстановки после определенного количества вложений используется директива

```
.IFCLEAR
```

Данная директива не выполняется, если находится внутри ассемблерного блока, который не обрабатывается в силу ложности условия. Она может использоваться в рекурсивном макросе для обработки сбалансированных пар директив IF - ENDIF, позволяя завершиться рекурсивным вызовам подстановки. При рекурсии она выполняет ту же функцию, которую выполняет директива MACEXIT в случае отсутствия рекурсии.

Поскольку сложная текстовая подстановка производится по имени, записанному в поле мнемокода, программисту важно знать, в каком порядке транслятор производит поиск мнемонических описаний. В первую очередь производится поиск в таблице мнемонических обозначений, задающих команды микроконтроллера, затем в таблице макроопределений (сложных текстовых подстановок). После этого следует поиск в таблице директив ассемблера и в последнюю очередь в таблице имен секций. Для переопределения порядка поиска может быть использована директива

```
.MACFIRST  ON
.MACFIRST  OFF
```

По включению приоритета сложных текстовых подстановок поиск будет осуществляться сначала в таблице макроопределений, а уже затем в таблице мнемонических обозначений.

2.7. Директивы управления вводом и выводом

При трансляции можно включить в исходный текст другой текстовый файл, ввод которого транслятором обеспечивается директивой

```
.INCLUDE    файл
```

По этой директиве указанный файл включается в заданное место исходного текста до начала трансляции. Если файл находится в одном каталоге с транслятором, то достаточно указать имя файла и его расширение. В противном случае нужно также добавить в операнд путь к файлу. Вложенное включение файлов не допускается.

При трансляции можно управлять выдачей листинга. Для управления шириной строки и количеством строк на странице, выдаваемой на принтер, предназначены директивы

```
.PW ширина
.PL высота
```

По умолчанию строка содержит 132 символа, а страница содержит 61 строку. При достижении заданной границы по ширине ассемблер выдает код перевода строки. При достижении заданной границы по высоте ассемблер выдает код перевода страницы. Если обнаруживается ошибка, то после выдачи сообщения об ошибке ассемблер выдает код перевода страницы. Страницы нумеруются. Для управления числом строк от верхней кромки страницы до номера страницы используется директива

.TOP *число*

По умолчанию используется значение 0.

На каждой странице листинга можно выводить заголовок и строкой ниже подзаголовок при помощи директив

.TITLE *строка*
.SUBTITLE *строка*

Строка печатается у верхнего края каждой страницы. Если строка не задана (пустой операнд), то действие предыдущей директивы с таким же мнемокодом отменяется. Заголовок и подзаголовок могут изменяться произвольное число раз независимо друг от друга. Они могут быть отменены в любой момент времени. Длина строки не должна превышать 80 символов. Кроме того, первые два символа табуляции между самой директивой и началом строки (если они существуют) будут игнорироваться. Все расположенные далее символы пробела и табуляции будут включаться в заголовок. Синонимами директивы вывода заголовка являются NAM, TTL и HEADING. Синонимами директивы вывода подзаголовка являются STTL и SUBTTL.

Для перевода страницы до завершения вывода заданного количества строк можно выдать директиву

```
.PAGE
```

Синонимами этой директивы являются PAG и EJECT.

Кроме управления форматом печати можно отключать вывод в листинг некоторой части протокола трансляции или включать его. В некоторых случаях одна строка исходного текста транслируется в несколько строк объектного кода. Чтобы уменьшить размер листинга, можно не печатать дополнительные строки объектного кода. Вывод на печать всех строк объектного кода задается директивой

```
.ASCLIST ON
```

Этот режим является стандартным (используемым по умолчанию). Вывод на печать только одной строки объектного кода для каждой строки исходного текста задается директивой

```
.ASCLIST OFF
```

Можно полностью отключать и снова включать вывод результатов трансляции в листинг директивами

```
.LIST OFF
.LIST ON
```

При запуске программы подразумевается, что задана директива LIST ON. Синонимом директивы отключения вывода являются NOLIST и NLIST, а синонимом директивы включения вывода — LIST. Аналогичным образом можно управлять выводом в листинг результатов трансляции сложных текстовых подстановок (макросов):

```
.MACLIST OFF
.MACLIST ON
```

Стандартным является режим раскрытия текста подстановок. Синонимом директивы отключения вывода подстановок является MNLIST, а синонимом директивы включения вывода подстановок — MLIST. Для управления выводом тех частей исходного текста, которые не транслируются в соответствии с условными директивами, используются директивы

```
.CONDLIST OFF
.CONDLIST ON
```

Стандартным является режим включенной генерации листинга ассемблерных блоков, которые не ассемблируются в силу ложности условия.

Обычно протокол трансляции выводится в листинг при втором проходе. Но при выдаче сообщения 'Symbol value changed between passes' об изменении при втором проходе адреса, приписываемого имени, требуется выяснить причину ошибки. Для управления выводом результатов первого прохода транслятора в листинг используются директивы

```
.PASS1 ON
.PASS1 OFF
```

Данные директивы могут также оказаться полезными для обнаружения ошибок, возникающих во вложенных ассемблерных блоках, которые ассемблируются в зависимости от выполнения условий. Они могут быть использованы в качестве вспомогательного средства для поиска ошибок, возникающих из-за различных маршрутов обработки, которые ассемблер реализует при выполнении прохода 1 и прохода 2.

При необходимости общения с программистом во время трансляции можно использовать директиву запроса

```
имя: ASK запрос
```

По этой директиве на дисплей выдается текст запроса, в ответ на который программист должен нажать цифровую клавишу. Из кода введенного символа вычитается 30h и полученное число присваивается имени, указанному в поле метки. Это значение может быть использовано в качестве аргумента директивы условной трансляции. Нажатие клавиши “Enter” без ввода цифры вызывает повторный запрос.

Программист может управлять выдачей сообщений о катастрофических ошибках, вставляя в условные директивы директиву вывода сообщения на дисплей

```
.EXIT "сообщение"
```

Длина задаваемого программистом сообщения не должна превышать 79 символов. После вывода сообщения работа транслятора завершается.

Существуют директивы, которые позволяют управлять компоновкой объектных файлов. При записи исполняемых файлов в ПЗУ может понадобиться заполнение пустых мест между специальными кодами, отличными от 0. В этом случае используется директива

.FILLCHAR *код*

Компоновщик будет заполнять пропуски, которые создаются при использовании секций или начал блоков, заданным значением. Для получения действительных адресов в исполняемых командах используется директива

```
.LINKLIST
```

Эта директива выполняется только совместно с опцией записи на диск. По этой директиве редактор связей преобразует ассемблерные листинги таким образом, что выполняемый адрес, адреса в полях кода объектных модулей и значения в таблице перекрестных ссылок представляют собой фактические значения в период выполнения. Для выбора опций редактора связей используется директива

.OPTIONS *опции*

В опциях можно указать выходной формат файла. По умолчанию выходной файл имеет шестнадцатеричный формат фирмы Intel. Можно заказать запись на дисковый файл карты памяти, списка глобальных имен и ошибок компоновки. Опции могут быть заданы при работе с компоновщиком в диалоговом и командном режимах или под управлением из файла. Информация об опциях приведена в описании диалогового режима работы с компоновщиком. Директива выбора опций выполняется только тогда, когда они не заданы при вызове компоновщика. Для изменения длины записей в таблицах имен может использоваться директива

```
.RECSIZE  значение
```

Указанное значение принимает длина записи выходного файла вместо 32 байт данных формата фирмы Intel или вместо 131 байт данных формата S фирмы Motorola. Компоновщик может создавать файлы с несколькими различными типами таблиц имен. Эти форматы поддерживают десятибуквенные и тридцатидвухбуквенные глобальные имена. Для размещения таблицы имен в выходном файле редактора связей нужно использовать директиву

```
.SYMBOLS ON
```

Она обеспечивает вывод таблицы в форматах фирм Microtek и Zax.

Глава 3

Кросс-средства фирмы 2500 A.D. Software, Inc. для семейства i8051

3.1. Общие сведения по пакету программ

В отличие от программирования на компьютерах, где все программы работают в одной и той же системе команд, технология программирования для микроконтроллеров другая. Ввиду ограниченных ресурсов микроконтроллерных систем исходные тексты их программ создаются и обрабатываются в персональном компьютере вплоть до получения исполняемой программы. В этом случае говорят, что инструментальная система другая, а ассемблирование называют кросс-ассемблированием (перекрестным ассемблированием). Существуют компьютерные программы, имитирующие работу микроконтроллеров, что может быть весьма полезно для отладки программ на инструментальных компьютерах. Такие методы моделирования работы программ называются математическими. При помощи инструментальных компьютеров и специальных приставок к ним осуществляется отладка программ в условиях, приближенных к реальной работе микроконтроллера с реальным изделием. Такие методы отработки называются эмуляцией. Инструментальные компьютеры используются и для записи исполняемого файла в ПЗУ при помощи периферийного устройства, называемого программатором. В данной главе приведены сведения о кросс-средствах, обеспечивающих создание программ для микроконтроллеров семейства i8051.

Пакет программ фирмы 2550 A.D. Software, Inc. предназначен для работы с исходными текстами программ, написанными на Ассемблере. Он пригоден для программирования большой группы процессоров семейства i8051 (i8080, i8085, z80, i8048 и др.). В его состав входят программы транслятора (ассемблера), редактора связей (компоновщика) и библиотекаря, работающие на персональных компьютерах, совместимых с IBM PC. Пакет пригоден для работы с операционными системами UNIX, VMX и MS DOS. Далее приведена информация по работе с этими программами только для MS DOS.

Читателей может заинтересовать вопрос, почему автор выбрал именно эти программы (с копирайтом 1985 года!), а не более современные? Ведь существуют аналогичные пакеты программ, работающие в интегрированной среде с меню или даже с пиктограммами. Нет сомнения, что работа с программами, имеющими современный интерфейс, более удобна. Но предметом нашей книги является изучение Ассемблера для микроконтроллеров, а не изучение интерфейсов инструментальных программ. К тому же программы наиболее свежей «выпечки» могут таить в себе ошибки, а старые программы выверены и отработаны.

В главе приведены общие сведения по трем утилитам: транслятору (ассемблеру) **X8051.EXE**, библиотекарю **LIB.EXE**, и компоновщику (редактору связей) **LINK.EXE**. Программы пакета используют по умолчанию следующие расширения имен файлов:

- ASM — входной файл для транслятора,
- OBJ — выходной файл из транслятора и входной файл для редактора связей и библиотекаря,
- PAK — упакованный выходной файл из транслятора и входной файл для библиотекаря,
- LST — выходной файл из транслятора,
- LIB — выходной из библиотекаря и входной файл для редактора связей,
- TSK — выходной файл из редактора связей в формате исполняемого машинного кода,
- HEX — выходной файл из редактора связей в формате фирмы Интел.

Отметим, что ассемблером включается в объектный файл дополнительная информация для компоновки модулей. Поэтому для получения исполняемого машинного кода объектный файл должен быть обработан редактором связей, даже если программа размещена с требуемого адреса и не

содержит внешних ссылок. При этом удаляется дополнительная информация и генерируется файл в нужном формате.

При установке пакета программ следует иметь в виду, что для работы программы-библиотекаря требуется загрузка ANSI-драйвера. Для этого необходимо включить в файл CONFIG.DYDS следующую строку:

```
DEVICE=ANSI.SYS
```

Далее описаны различные способы работы с каждой из утилит и приведены сведения о выдаваемых ими сообщениях (о выявленных ошибках).

Две из перечисленных утилит (библиотекарь и компоновщик) предназначены для более широкого применения. Они могут работать с объектными файлами, полученными в результате трансляции с диалектов Ассемблера, относящихся к процессорам с другой архитектурой и системой команд. Формат объектных файлов, с которыми работают утилиты пакета, несовместим с форматом объектных файлов фирмы Майкрософт. Компоновщик может выдавать исполняемые файлы в различных форматах, но для их загрузки в микроконтроллеры семейства i8051 обычно используются машинный код и формат фирмы Интел.

3.2. Работа с транслятором

Работать с транслятором можно в режиме диалога и в режиме командной строки. Для работы в режиме диалога нужно набрать на клавиатуре имя программы, а именно x8051, и нажать клавишу “Enter”. Далее запросы и сообщения утилит будут выделяться подчеркиванием. Транслятор в ответ на вызов выдает заголовок и запрос о том, куда направить листинг:

```
         8051 Macro Assembler   -   Version 4.04a
      Copyright (C) 1985 by 2500 A.D. Software, Inc.
Listing Destination (N, T, D,E ,L , P,<CR> = N): (Вывод листинга)
```

где аббревиатуры означают следующее:

N — листинг не нужен, по умолчанию листинг также не выдается,
T — вывод листинга на терминал,
P — вывод листинга на принтер,
D — вывод листинга на диск,
E — вывод только сообщений об ошибках,
L — разрешение управления выводом по директивам LIST ON/OFF.

В последнем случае выдается дополнительный запрос:

LIST ON/OFF Listing Destination (T, ,D,<CR>=T): (Вывод листинга)

Сокращения соответствуют предыдущим. Если заказана выдача только сообщений об ошибках, то транслятор запрашивает, куда их выводить:

Error Only Listing Destination (T,P,D,<CR>=T): (Вывод ошибок)

Если листинг выводится на принтер (для однопользовательских систем) или на диск, то выдается запрос о включении в листинг таблицы ссылок:

Generate Cross Reference (Y/N <CR> = No): (Вывод таблицы ссылок)

Если листинг не нужен, то этот запрос не выдается. Затем транслятор запрашивает имя файла, содержащего исходный текст:

Input Filename: (Имя входного файла)

При вводе имени файла расширение ASM можно опустить. После ввода имени входного файла транслятор запрашивает имя выходного файла:

Output Filename: (Имя выходного файла)

По умолчанию имя выходного файла определяется именем входного с расширением OBJ. Если же умалчивается только расширение, то в имени выходного файла принимается расширение OBJ. После этого транслятор выводит информацию о возможности управления транслятором при помощи клавиатуры:

***** Active Commands *****	Управление с клавиатуры
Ctrl S = Stop Output	Приостановить вывод
Ctrl Q = Start Output	Возобновить вывод
Esc C = Stop Assembly	Прекратить ассемблирование
Esc T = Terminal Output	Вывод на дисплей
Esc P = Printer Output	Вывод на принтер
Esc D = Disk Output	Вывод на диск
Esc M = Multiple Output	Вывод на все устройства
Esc N = No Output	Прекратить вывод

Перечисленные команды можно выдавать с клавиатуры во время работы транслятора как при первом, так и при втором проходах. Если вывод информации на терминал не отменялся, то в процессе трансляции выдаются сообщения об ошибках и строки исходного текста, в которых обнаружены эти ошибки. По окончании процесса трансляции выдается сообщение о результатах обработки исходного текста с именами исходного и объектного файлов, с количеством обработанных строк и выявленных синтаксических ошибок:

```
2500 A.D. 8051 Macro Assembler  -  Version 4.04a
------------------------------------------------
```

```
        Input  Filename : имя.asm
        Output Filename : имя.obj
Lines Assembled :  xx             Assembly Errors : yy
```

Буквами xx и yy здесь обозначены количество строк, обработанных ассемблером, и количество выявленных ошибок соответственно.

При работе в режиме командной строки нужно сообщить транслятору ту же самую информацию, что и в режиме диалога. Синтаксис команды соответствует стандартам операционной системы UNIX, то есть дефис означает, что за ним следует опция. Общая форма команды (необязательные поля показаны в квадратных скобках) будет следующей:

```
x8051[-q]input_filename[output_filename][-t,-p,-d,-px,-dx]
```

Имя входного файла определяется первым, дополнительно можно ввести имя выходного файла и список опций. По опции q, предшествующей имени файла, на экран выводятся только сообщения об ошибках и строки исходного текста, в которых обнаружены эти ошибки. Остальные опции предназначены для управления выводом листинга:

- t — вывод листинга на терминал
- p — вывод листинга на принтер
- x — добавление таблицы перекрестных ссылок
- d — вывод листинга на диск
- e — вывод только сообщений об ошибках
- l — разрешение управления выводом по директивам LIST ON/OFF.

В режиме командной строки транслятор после завершения работы выводит на терминал такое же сообщение, как в режиме диалога.

3.3. Сообщения транслятора об ошибках

Независимо от выбранного режима работы транслятор проверяет исходный текст и выдает сообщения о синтаксических ошибках (в случае их обнаружения). Сообщения о чтении транслятором символов, не входящих в алфавит Ассемблера, не выдаются. В этом случае на экран или в файл листинга выводится только номер строки исходного текста и какой-либо числовой код. Однако этот вид ошибок учитывается при подсчете их общего количества.

Следует обратить особое внимание на существенную разницу в смысле двух сходно звучащих терминов: русского «символ» и английского «symbol». Первый из них означает некоторый отображаемый на дисплее или на бумаге значок (на английском character). Второй термин соответствует символическому обозначению программного объекта. Когда американский программист говорит или пишет “symbol”, это обозначает имя, а не букву или цифру!

Если вывод листинга отменен, то на терминал все равно выводятся строки исходного текста, в которых обнаружены ошибки, и сообщения о типах ошибок. Приостановить и возобновить вывод можно нажатием пар клавиш Ctrl + S и Ctrl + Q соответственно. Приостановка вывода дает программисту возможность прочитать строку исходного текста, явившуюся источником сообщения об ошибке (вообще говоря, источник ошибки может находиться в другом месте!). Ошибка может быть обнаружена и на первом и на втором проходе. Если назначен вывод листинга на принтер или диск, то сообщения об ошибках, обнаруженных при первом проходе, выводятся только на терминал и ассемблирование не прекращается. При втором проходе ошибки выводятся на терминал параллельно с выводом на принтер и диск. Впрочем, специальной директивой в исходном тексте можно заказать вывод в листинг сообщений об ошибках при первом проходе. Ошибки, обнаруженные при втором проходе, выводятся на терминал и в листинг, даже если ошибка находится в блоке, не предназначенном для вывода в листинг. Далее в алфавитном порядке перечислены сообщения транслятора об ошибках с пояснениями на русском языке.

<u># TOO LARGE</u> Слишком большое число.

<u>ACCUMULATOR OR R0 - R7 REQUIRED</u> Операндом должен быть накопитель или регистр общего назначения.

<u>A LABEL IS ILLEGAL ON THIS INSTRUCTION</u> Использование метки в данной команде недопустимо.

<u>ATTEMPTED DIVISION BY ZERO</u> Попытка деления на ноль в выражении.

<u>BRANCHES MUST BE WITHIN CURRENT SECTION</u> Переходы должны быть в пределах текущего сегмента.

<u>CAN'T CREATE OUTPUT FILE - DISK MAY BE FULL</u> Невозможно создание выходного файла, может быть, недостаточно места на диске. (Если места на диске достаточно, то нужно увеличить количество файлов, открываемых операционной системой.)

<u>CAN'T FIND FILE: NAME.EXT</u> Файл с таким именем и расширением не обнаружен. (Такого файла не существует либо операционная система не допускает одновременного открытия необходимого количества файлов.)

<u>CAN'T OPEN INPUT FILE</u> Невозможно открытие входного файла.

(Операционная система не допускает одновременного открытия необходимого количества файлов.)

CANT RECOGNIZE NUMBER BASE Транслятор не может опознать основание системы счисления.

CAN'T RESOLVE OPERAND Транслятор не может обработать операнд.

CODE AND DATA GENERATION NOT ALLOWED IN CURRENT SECTION В текущем сегменте не разрешается записывать программу и данные.

DESTINATION ADDRESS NOT IN SAME 2K BLOCK AS NEXT INSTRUCTION Адрес перехода вне пределов 2-хкилобайтового блока относительно адреса следующей команды.

DIRECTIVE DETECTED AT END OF ASSEMBLY По окончании трансляции обнаружена директива.

'ENDM' OR 'MACEND' OUTSIDE OF A MACRO Закрывающая директива сложной текстовой подстановки вне ее определения.

'ENDMOD' CAN'T BE IN 'INCLUDE' FILE Использование директивы ENDMOD во включаемом файле не допускается.

EXTRA CHARACTERS AT END OF OPERAND Лишние символы в конце операнда.

HEX # AND SYMBOL ARE IDENTICAL Совпадение имени с шестнадцатеричным числом.

ILLEGAL ADDRESSING MODE Недопустимый режим адресации.

ILLEGAL ASSIGNMENT Недопустимое назначение места в памяти.

ILLEGAL ASCII CHARACTER Недопустимый символ ASCII.

ILLEGAL DIRECTIVE IN MICROSOFT REL FORMAT Недопустимая директива для относительного формата Microsoft.

ILLEGAL EXTERNAL REFERENCE Недопустимая внешняя ссылка.

ILLEGAL LABEL 1ST CHARACTER Недопустимый первый символ в поле метки. (Имя должно начинаться с буквы.)

ILLEGAL LOCAL LABEL Недопустимое использование локальной метки.

ILLEGAL MNEMONIC Недопустимый мнемокод

ILLEGAL NESTED INCLUDE Недопустимое включение файла во включаемый файл. (Данное сообщение об ошибке может также указывать на отсутствие оператора END в каком-либо включаемом файле.)

ILLEGAL REGISTER Недопустимое обозначение регистра.

LABEL VALUE CHANGED BETWEEN PASSES Значение метки изменилось между первым и вторым проходами транслятора. (Эта ошибка обычно возникает в том случае, когда транслятор обрабатывает разные части программы при выполнении двух проходов из-за изменения значений аргументов директив условной компиляции.)

LABEL WAS NOT DEFINED ON PASS 1. (MIGHT CAUSE OTHER ERRORS)

Метка не определена на первом проходе (может стать причиной других ошибок).

MACRO NAME MUST APPEAR ON SAME LINE AS MACRO DEFINITION Имя сложной текстовой подстановки должно быть записано в той же строке, где начинается ее описание.

MACRO STACK OVERFLOW Переполнение стека сложных текстовых подстановок (макроопределений). (Данная ошибка может быть вызвана слишком большим количеством рекурсивных вызовов подстановки. Стек допускает использование приблизительно 700 вложенных вызовов. Допустимое количество вложений зависит от количества аргументов, используемых в подстановках.)

MAXIMUM EXTERNAL SYMBOL COUNT EXCEEDED Превышено максимальное количество внешних имен. (Максимальное количество внешних имен в модуле около 500.)

MISSING DELIMITERS ON MACRO CALL LINE В строке вызова подстановки пропущены ограничители.

MISSING 'ENDM' OR 'MACEND' Пропущена закрывающая директива сложной текстовой подстановки.

MISSING 'ENDMOD' DIRECTIVE Пропущена директива ENDMOD (конец модуля).

MISSING LABEL Пропущена метка.

MISSING 'MODULE' DIRECTIVE Пропущена директива MODULE (начало модуля).

MISSING RIGHT ANGLE BRACKET Пропущена правая угловая скобка.

'MODULE' CAN'T BE IN 'INCLUDE' FILE Использование директивы MODULE во включаемом файле не допускается.

MULTIPLY DEFINED SYMBOL Повторное определение имени (в поле метки).

MULTIPLE EXTERNALS IN THE SAME OPERAND Одному и тому же операнду соответствуют несколько внешних имен.

MUST BE IN SAME SECTION Операнд команды должен находиться в той же самой секции.

NESTED CONDITIONAL ASSEMBLY UNBALANCE DETECTED Несоответствие количества открывающих и закрывающих строк директивы условной трансляции.

NESTED SECTION UNBALANCE Несоответствие количества открывающих и закрывающих строк директивы вложения секций. (В описании вложенной секции отсутствует ENDS.)

NOT ENOUGH PARAMETERS Недостаточное количество параметров.

NON-EXISTENT INCLUDE FILE Включаемый файл не найден.

NON-EXISTENT INTERNAL RAM ADDRESS Несуществующий адрес внутреннего ОЗУ.

OPERAND MUST BE DEFINED AS AN 8 BIT RELOCATABLE VALUE

Операнд должен быть описан как 8-битовое перемещаемое значение. (Это сообщение об ошибке возникает при попытке использования 16-битового адреса в команде, работающей с 8-битовым адресом. Для того чтобы сделать это значение перемещаемым, необходимо выделить байт операцией < или >.)

ORG'S CANNOT BE USED ALONG WITH SECTION INDIRECT MODIFIERS
Нельзя использовать директиву записи в счетчик адреса при косвенном задании сегмента.

ORG'S CANNOT BE USED ALONG WITH SECTION OFFSET MODIFIERS
Нельзя использовать директиву записи в счетчик адреса при задании сегмента со смещением.

R0 OR R1 REQUIRED Необходимо использовать 0-ой или 1-ый регистр общего назначения.

REGISTER IS NOT BIT ADDRESSABLE Регистр не допускает битовую адресацию.

RELATIVE JUMP TOO LARGE Передача управления с относительной адресацией за пределы от -128 до +127.

SYNTAX ERROR Синтаксическая ошибка. (Обычно это сообщение выдается из-за пропущенной запятой или круглой скобки.)

THIS REGISTER HAS MORE THAN 1 VALUE Этот регистр имеет более одного адреса. (Использование регистровой или косвенной адресации для этой команды не допускается.)

UNDEFINED SYMBOL Имя не определено в поле метки.

UNEXPECTED END OF MACRO DETECTED Обнаружено неожиданное окончание определения сложной текстовой подстановки.

Следует иметь в виду, что в ряде случаев одна ошибка может привести к выводу транслятором нескольких сообщений об ошибке.

3.4. Работа с библиотекарем

Библиотекарь предназначен для создания и обслуживания библиотечных файлов, состоящих из заданных пользователем объектных модулей. Эти модули могут быть использованы компоновщиком при создания исполняемых программ вместе с объектными файлами, не входящими в библиотеку. Компоновщик осуществляет поиск в заданных библиотечных файлах и присоединяет только те модули, на которые имеется ссылка. Библиотечный файл может содержать до 256 модулей. Библиотекарь обрабатывает файлы любого размера при условии, что в свободном дисковом пространстве помещается временный файл, размер которого равен сумме размеров существующего библиотечного файла и добавляемых к нему модулей. Временный файл используется для создания промежуточного библиотечного файла в целях минимизации возможного повреждения

существующего. Программе библиотекаря также требуется наличие достаточной памяти для размещения списка модулей и всех записей глобальных символов этих модулей. Библиотекарь проверяет уникальность глобальных имен в пределах библиотечного файла. Библиотечные файлы имеют расширение LIB, а присоединяемые к ним объектные модули должны иметь расширение OBJ или входить в состав файлов с расширением PAK. Модулем может являться единственный объектный файл или модуль в упакованном объектном файле. Упакованный объектный файл создается при использовании директив MODULE и ENDMOD и содержит несколько соединенных вместе модулей. Упакованный объектный файл может использоваться для выполнения каких-либо операций над отдельным модулем или всеми модулями в этом файле.

Программа-библиотекарь может работать только в режиме командной строки. Для ее вызова необходимо ввести команду

`lib` *имя*

При помощи имени может быть указан только один библиотечный файл (уже существующий или вновь создаваемый). Эта команда не имеет никаких опций. Библиотекарь работает в интерактивном режиме, то есть выполняет вводимые с клавиатуры команды до тех пор, пока не будет выдана команда выхода, по которой управление возвращается операционной системе. Поэтому его команды не могут быть использованы в пакетном файле операционной системы MS DOS.

Если при вызове библиотекаря было указано имя существующего библиотечного файла, то на экране отображается список имен содержащихся в нем модулей. На экране может быть отображено до 16 имен. Имя текущего (рабочего) модуля выделяется с помощью светового маркера. Текущий модуль можно сменить посредством прокрутки строк списка вверх или вниз. Для прокрутки вверх необходимо нажать клавишу k или K, а для прокрутки списка вниз необходимо нажать клавишу j или J. Если к списку добавляется новый модуль, который оказывается вне экранного кадра с отображаемыми в текущий момент именами, то происходит сдвиг отображаемого списка, в результате чего отображается имя добавленного модуля. При добавлении модуля его имя всегда отображается как текущий модуль. При умолчании имени в команде вызова библиотекаря можно задать имя файла при помощи команды NEW. В нижней строке (независимо от вывода списка или размера выведенного списка) библиотекарь выдает запрос следующего вида:

`Enter Command:` (Введите команду)

В тех случаях, когда после исполнения команды на экран выводится другая информация, для перехода к вводу очередной команды нужно нажать любую клавишу. Библиотекарь сообщает об этом выводом строки

Press Any Key To Continue. (Для продолжения нажмите любую клавишу)

Описание команд библиотекаря приведено ниже в алфавитном порядке. В описаниях команд приведены полное и сокращенное имя команды, а также операнды, необходимые для их работы.

Чтобы добавить модуль к библиотеке или заменить существующий модуль в библиотеке, используется команда

ADD *имя*

Сокращенная запись команды — **A**. Использование этой команды разрешено только тогда, когда открыт существующий или вновь создаваемый файл библиотеки. Операндом команды является имя добавляемого модуля или слов **all** или **ALL**, используемых для добавления всех модулей упакованного объектного файла. Библиотекарь запрашивает имя объектного файла, содержащего добавляемый модуль:

Enter Name of Object File (Введите имя объектного файла)

Если добавляется отдельный модуль, то библиотекарь сначала ищет файл с расширением PAK, а если не найдет такового, то ищет файл с расширением OBJ. Найденный модуль включается в библиотеку, если количество модулей не достигло 256. Если команду не удается выполнить, то библиотекарь выдает сообщение об ошибке.

Для перемещения маркера в конец списка модулей используется команда

BOT

Эта команда не требует операнда.

Для удаления модуля используется команда

DEL *имя*

Сокращенная запись команды — **D** .Команда удаляет из библиотеки заданный модуль или (по умолчанию) текущий модуль, имя которого выделено световым маркером. Имя удаленного модуля удаляется из списка, выводимого на экран, а текущим модулем становится или следующий по списку, или предыдущий, если был удален последний модуль в списке. Если команду не удается выполнить, то библиотекарь выдает сообщение об ошибке.

Для окончания работы с библиотекарем с сохранением результата работы используется команда

EXIT

Если открытый файл содержит хотя бы один модуль, то по этой команде осуществляется его запись на диск и возврат управления операционной системе. При сохранении файла программа выдает сообщение

Saving Library: имя (Сохранение библиотеки)

В сообщении указывается имя сохраняемого файла.

Для поиска модуля используется команда

FIND *имя*

Она помогает установить световой маркер на заданное имя в списке модулей библиотечного файла. Если указанный модуль не найден, то выдается сообщение об ошибке.

Для получения справок по командам библиотекаря используется команда

HELP строка

Сокращенная запись команды — **H.** Команда HELP по умолчанию операнда отображает сводную информацию по всем командам программы-библиотекаря или описание той команды, которая задана операндом. Если в качестве операнда используется название команды, то выдается подсказка по правилам использования команды. На время работы команды на экране вместо списка модулей отображается справочный текст. Библиотекарь требует нажатия клавиши для продолжения работы:

Press Any Key To Continue.

После нажатия клавиши библиотекарь возобновляет вывод списка модулей на экран.

Для вывода перечня глобальных имен на экран или в файл используется команда с двумя операндами

LIST модуль файл

Вместо имени модуля можно ввести слова **all** или **ALL**, по которым осуществляется вывод перечней глобальных символов из всех модулей библиотеки. Имена в перечне располагаются в алфавитном порядке. По умолчанию второго операнда перечень выводится на экран. Для записи перечня на диск во втором операнде должно быть записано **disk** или **DISK**.

Если имя файла не было задано при вызове библиотекаря, то для открытия существующего файла или вновь создаваемого файла используется команда

NEW *имя*

Сокращенная запись команды — **N**. По отсутствию операнда библиотекарь запросит ввод имени файла. Если открытый ранее файл содержит по меньшей мере один модуль, то он сохраняется перед тем, как открыть новый файл. Добавление, удаление или замена модулей в библиотеке станут возможными только тогда, когда открыт файл. При открытии существующего файла на экран выводится список модулей.

Для выхода из библиотекаря без сохранения результата работы используется команда

```
QUIT
```

Если перед этим был открыт файл, содержащий по меньшей мере один модуль, то библиотекарь выдает запрос о необходимости его сохранения:

`Do You Want to Save the Library (y/n)` `(Хотите ли вы сохранить библиотеку)`

При вводе литеры **y** или **Y** файл сохраняется, а ввод любого другого символа приведет к потере внесенных изменений при выходе из библиотекаря.

Для замены существующего модуля на новую версию используется команда

`REP` *`имя`*

Сокращенная запись команды — **R**. В операнде должно быть указано имя заменяемого модуля или слова **all** или **ALL**, используемые для замены всех модулей (только при работе с упакованными объектными файлами). По умолчанию операнда производится замена текущего модуля. Если модуль с заданным именем не найден, то выдается сообщение об ошибке. Для поиска новой версии модуля выдается запрос

`Enter Name of Object File` `(Введите имя объектного файла)`

По введенному имени библиотекарь ищет модуль сначала в файле с расширением OBJ, а если модуль не найден, то в файле с расширением PAK. Если новый модуль найден, то осуществляется замена старого модуля на новый. Если команду не удается выполнить, то библиотекарь выдает сообщение об ошибке.

Для отображения статуса текущего файла используется команда

```
STAT
```

Сокращенная запись команды — **S**. Если файл не открыт, то на экран выдается сообщение об отсутствии открытого файла. Если файл открыт, то на экран выдается его имя и общее количество модулей, внешних и глобальных имен. При этом с экрана удаляется список модулей и отображается сообщение о статусе файла. Затем библиотекарь предлагает для продолжения работы нажать какую-либо клавишу:

Press Any Key To Continue.

После нажатия клавиши на экране возобновляется показ списка модулей.

Для перемещения светового маркера на первое имя в списке модулей используется команда

```
TOP
```

Эта команда не требует операнда.

3.5. Сообщения библиотекаря об ошибках

Ниже приводится упорядоченный по алфавиту перечень сообщений об ошибках, выводимых программой-библиотекарем, и объяснение по каждому сообщению об ошибке.

Cannot Create New Library Невозможно создание новой библиотеки. (Временный библиотечный файл не может быть переименован. Вместе с сообщением об ошибке выводится имя файла.)

Cannot Delete Old Library Невозможно удаление старой библиотеки. (Удаление старого файла библиотеки после обновления или создания нового файла оказалось невозможным. Вместе с сообщением об ошибке выводится имя файла.)

Cannot Open File Невозможно открытие файла. (Невозможно обращение к заданному файлу или запись в него.)

Filename Too Long Слишком длинное имя файла. (После того как было добавлено стандартное расширение, специфицированное имя файла оказалось слишком длинным. Вместе с сообщением об ошибке выводится имя файла.)

Illegal Command Недопустимая команда. (Введенная команда не относится к числу допустимых команд или команда ADD, DEL или REP была использована, когда библиотечный файл не был открыт.)

Incompatible Object Module Несовместимый объектный модуль. (В качестве аргумента для команды ADD был задан библиотечный файл или в качестве аргумента для команды NEW был задан объектный файл.)

Input Line Too Long Входная строка слишком длинна. (Длина входной строки не должна превышать 80 символов.)

Maximum Module Count Exceeded Исчерпано максимальное число модулей. (Количество модулей в файле должно быть не более 256.)

Multiple Defined Global Symbol Повторно описано глобальное имя. (Глобальное имя в добавляемом модуле содержит какое-либо имя, уже существующее в другом модуле. Вместе с сооб-

щением об ошибке выводится глобальное имя и имя файла.)
Multiple Defined Module Повторно описан модуль. (Имя добавляемого модуля уже существует в этой библиотеке. Осуществите замену этого модуля или переименуйте модуль, который должен быть добавлен. Вместе с сообщением об ошибке выводится имя файла.)
Must Be A Packed Object File Нужен упакованный объектный файл. (В команде ADD или REP был использован операнд "all" или "ALL", а заданный файл не является упакованным объектным файлом.)
Not Enough Memory Недостаточно памяти. (Недостаточно памяти для буфера или для таблицы имен.)
Read Error Ошибка чтения. (Ошибка возникла при чтении файла. Вместе с сообщением об ошибке выводится имя файла.)
Seek Error Ошибка поиска. (Ошибка возникла при поиске в обновляемом файле.)
Too Many Command Line Arguments Слишком много аргументов задано в командной строке. (Библиотекарь был вызван из командной строки операционной системы с двумя или более аргументами.)
Undefined Module Неопределенный модуль. (Удаляемого или замещаемого модуля нет в библиотеке.)
Write Error Ошибка записи. (Ошибка возникла при обновлении файла. Вместе с сообщением об ошибке выводится имя файла.)

3.6. Работа с компоновщиком (редактором связей)

Компоновщик предназначен для создания машинного кода исполняемой программы с целью последующей его записи в ПЗУ микроконтроллерного устройства. Для создания исполняемой программы он использует объектные файлы в формате, созданном транслятором, и библиотечные файлы, сформированные библиотекарем из объектных файлов. Таким образом компоновщик позволяет программисту писать исходный текст в виде нескольких модулей на Ассемблере. Компоновщик выполняет размещение программ и данных в адресном пространстве соответствующих сегментов (секций) и учитывает внешние ссылки. Он способен создавать исполняемые файлы нескольких форматов, применяемых для загрузки в ПЗУ.

Во время создания выполняемого файла вся программа редактора связей находится в оперативной памяти. Для составления исполняемого файла компоновщик создает столько помеченных файлов, сколько программных секций требуется скомпоновать (с учетом их сортировки). Каждый объектный файл может иметь до 256 различных секций, включая

определенные программистом. Для разрешения внешних ссылок компоновщик может работать с 50 различными библиотечными файлами. Всего компоновщик может работать в комбинации с 256 входными файлами и библиотечными модулями и 256 различными именами секций. На размеры секций ограничения не накладываются. Файлы могут компоноваться в соответствии с адресами, указанными в самом файле или определяемыми в момент компоновки. Регистровые и битовые секции могут быть использованы только для ссылок. Информация этих секций необходима для процесса компоновки и не включается в выходной файл. Если объем памяти и диска достаточен для размещения файлов, то процесс компоновки проходит нормально.

Машинный код записывается в выходной файл, который может быть выдан в одном из двух форматов: исполняемый двоичный файл (расширение TSK) или шестнадцатеричный формат фирмы Intel (расширение HEX). Кроме машинного кода компоновщик может создавать дополнительные файлы в соответствии с директивами и опциями на момент исполнения компоновки (последние имеют приоритет перед директивами). В процессе компоновки на экран выдаются предупреждения (в этом случае компоновка продолжается) и сообщения об ошибках, после чего компоновка прекращается.

Компоновщик можно вызвать для работы в диалоговом режиме, в режиме командной строки и в режиме использования файла компоновки. Чтобы запустить компоновщик в диалоговом режиме, нужно ввести команду

```
link
```

После этого компоновщик выдает сообщение о себе и запрашивает имя входного файла:

```
2500 A.D. Linker  Copyright (C) 1985 - Version 4.04a
Input   Filename : (Имя входного файла)
```

По умолчанию принимается расширение файла OBJ. После чтения этого файла компоновщик запрашивает начальный адрес каждой из секций, имеющих ненулевой размер. Запросы имеют вид

```
Enter Offset For 'CODE'                          :
Enter Offset For 'DATA'                          :
Enter Offset For 'RSECT'                         :
Enter Offset For 'BSECT'                         :
```

Если в объектном файле есть секции с именем, заданным программистом, то компоновщик выдает запросы и по этим секциям. Введенная в шестнадцатеричном коде величина определяет начальный адрес секции. По умолчанию (нажатие только клавиши "Enter") компоновщик присоединяет секцию

к предыдущей. При вводе отрицательного числа адреса в секции смещаются на абсолютное значение введенного числа, но компоновщик не включает эту секцию в выходной файл. При вводе точки с запятой после любого введенного числа секция компонуется как продолжение предыдущей. После ввода всех смещений компоновщик вновь запрашивает имя объектного файла до тех пор, пока не будет введено пустое имя объектного файла. Затем компоновщик запрашивает имя выходного файла:

Output Filename :

По умолчанию выходному файлу присваивается имя первого объектного файла и расширение, соответствующее задаваемому далее выходному формату. После этого компоновщик запрашивает имя библиотечного файла:

Library Filename :

Компоновщик может обработать до 50 библиотечных файлов. Запросы продолжаются до тех пор, пока не будет введено пустое имя библиотечного файла. После этого компоновщик запрашивает формат выходного файла:

Options (D, P, S, A, M, N, Z, X, H, E, T, 1, 2, 3, <CR> = Default) :

Когда указано несколько конкурирующих параметров, последний параметр отменяет действие предыдущего. Вводимым буквам соответствуют следующие характеристики выходного файла:

- D — создающиеся в процессе компоновки дополнительные файлы записываются на диск. Эти файлы имеют то же имя, что и выходной файл, но другие расширения.
- P — создается файл с расширением MAP.
- S — создаются файлы с расширениями MAP и SYM (размер имени до 32 символов).
- A — создаются файлы с расширениями MAP и SYM (размер имени до 10 символов для совместимости с компоновщиком 2500 A.D. версии 3.0).
- M — создаются файлы с расширениями MAP и SYM (в формате Microtek). Для этого в исходном файле должна быть записана директива SYMBOLS ON.
- Z — создается файл с расширением SYM (с глобальными и локальными именами в формате ZAX). Для этого в исходном файле должна быть записана директива SYMBOLS ON.

- X — выполняемый выходной файл выдается с расширением TSK (для всех остальных опций — с расширением HEX).
- H — создается файл с расширением MAP.
- E — создается увеличенный файл с расширением HEX и файл с расширением MAP.
- T — создается выполняемый выходной файл с расширением TEK в формате Tektronix и файл с расширением MAP.
- 1 — создается выполняемый выходной файл с расширением S19 в формате фирмы Motorola.
- 2 — создается выполняемый выходной файл с расширением S28 в формате фирмы Motorola.
- 3 — создается выполняемый выходной файл с расширением S37 в формате фирмы Motorola.

По умолчанию формат выходного файла определяется директивой OPTIONS, а в случае ее отсутствия выдается выходной файл с расширением HEX.

После этого производится компоновка и выдаются сообщение о созданных в результате компоновки файлах:

```
Linker Output Filename : имя
Disk Listing  Filename : имя
Symbol Table  Filename : имя
Link Errors : количество     Output Format : тип
```

В той строке, для которой вывод файла не задан, вместо имени выдается

```
None Specified
```

Помимо работы в диалоговом режиме можно выдать задание компоновщику в командной строке.

При вызове компоновщика командной строкой необходимо задавать информацию в той же последовательности, что и в диалоговом режиме. Чтобы сообщить компоновщику о назначении информации, используются символы управляющей и информационной групп. Символы управляющей группы начинаются с дефиса, за ними могут следовать символы информационной группы.

- q – сообщает компилятору о режиме Quit (выход). В этом режиме компоновщик выдает только сообщение об ошибках на консоль.
- c – сообщает компилятору о режиме командной строки. За этим управляющим символом следуют имена объектных файлов, которые могут чередоваться с

символами управляющей и информационной групп для ввода значений смещения секций.

- I – сообщает компилятору о вводе смещения. За этим управляющим символом следуют цифровые символы, задающие величину смещения. Количество символов этого вида должно быть равно количеству секций объектного файла. Если они отсутствуют, то текущие секции являются продолжением предыдущих с тем же именем.
- o – сообщает компилятору об имени выходного файла. За этим управляющим символом следует имя выходного файла. При отсутствии строки такого вида компоновщик создает выходной файл с тем же именем, что и первый объектный файл, и с расширением, определяемым типом выходного файла.
- L – сообщает компилятору об именах библиотечных файлов. За этим управляющим символом следуют имена библиотечных файлов.

Формат такой команды показан ниже. Квадратными скобками (они не входят в состав вводимых символов!) обозначены необязательные группы символов командной строки.

link [-q]-s*имя*[-I*число*]*имя*[-I*число*]...[-o*имя*] [-L*имя*] [-*опции*]

Для того чтобы задать формат выходного файла, используются те же символы, что и в последнем запросе диалогового режима:

-опции - список дополнительных параметров.

Дефис нужен только в начале списка. В список можно включить любое количество параметров; разделители в списке не нужны. При работе с компоновщиком в диалоговом режиме и в режиме командной строки приходится вводить много информации с клавиатуры. Чтобы упростить работу с компоновщиком, используется ввод информации через файл компоновки.

Использование файла компоновки при вызове редактора связей является наиболее удобным. В файл компоновки необходимо записать в текстовом формате ответы на все запросы, которые компоновщик задает в диалоговом режиме. Файл компоновки должен иметь расширение LNK. При вызове компоновщика нужно указать имя файла компоновки

link *имя*

Каждая строка файла компоновки соответствует ответу на запрос. В случае умолчания можно ввести в файл пустую строку, но для удобства

программиста разрешается замена пустой строки символом подчеркивания. Наиболее простой способ создания файла компоновки начинается с пробной компоновки в диалоговом режиме с записью своих ответов на запросы программы. Затем нужно при помощи текстового редактора записать каждый из ответов в отдельной строке файла компоновки. В файл компоновки можно включать комментарии, которые записываются в отдельных строках и должны начинаться со звездочки (*) или точки с запятой (;).

3.7. Как вычисляются адреса при компоновке модулей

При компоновке программы из нескольких модулей компоновщик должен обеспечить их размещение в ПЗУ и скорректировать адреса таким образом, чтобы они могли работать без помех друг другу. С этой целью к адресам во всех секциях модуля, вычисленным при ассемблировании как относительные, компоновщик добавляет определенные для этих секций числа, называемые смещениями, Первая команда главного модуля программы должна быть записана по адресу 0000h, поэтому смещение секции кодов главного модуля должно быть равно нулю. Для остальных секций главного модуля и для всех секций других модулей программист должен задать смещения или положиться на самостоятельное вычисление смещений компоновщиком.

В процессе редактирования связей компоновщик ведет учет использованного адресного пространства ПЗУ и ОЗУ. По умолчанию каждая следующая секция добавляется к предыдущей без зазоров в адресном пространстве. Поэтому задавать смещения компоновщику нужно только в том случае, когда необходимо изменить последовательность расположения секций. Задавая смещения, программист должен точно знать объем адресного пространства всех секций компонуемых модулей, чтобы не нарушить их взаимодействия.

В этой связи следует отметить, что операнды машинных команд могут быть числами и адресами. Хотя и то и другое кодируется байтами, с точки зрения компоновки эти объекты обладают разными свойствами. Значение числового операнда, соответствующего непосредственному способу адресации, не зависит от расположения команды в адресном пространстве. Значение адресного операнда, соответствующего прямой адресации, зависит от расположения адресуемого объекта. При загрузке указателя в регистр для косвенной адресации также нужно позаботиться о том, чтобы его значение было адресным, а не числовым. При короткой

(или относительной) адресации команд операнд является числом, поскольку это разность адресов. При индексной адресации в регистр указателя данных должен быть записан адрес, а в накопитель — число, так как сумма адреса и числа является адресом. Суммирование адресов недопустимо.

При создании программы из нескольких модулей программист должен обеспечить их перемещаемость. Для проверки перемещаемости следует использовать таблицу имен. Ассемблер записывает в эту таблицу значения для каждого из имен, используемых в программе. Различие в значениях определяется операторами, в поле метки которых задано имя. Имя в поле метки команды порождает адрес. Имя в поле метки директивы EQU в случае числового операнда порождает число. Перед числовым значением в таблице имен выводится знак равенства, а перед адресным значением этого знака нет.

3.8. Сообщения компоновщика об ошибках

При невозможности выполнения компоновки выдается сообщение об ошибке, завершающееся информацией о прекращении компоновки:

Link Terminated (Компоновка прекращена)

Причины прекращения компоновки могут быть самыми разнообразными. В первую очередь нужно исключить возможность ограничения ресурсов компьютера и неподходящую среду для работы программы.

Corrupted environment Разрушена среда.

Invalid stack override directive Недопустимая директива переопределения стека.

LC 3.40 Incompatible DOS version Несовместимая версия DOS.

Not Enough Memory Недостаточно памяти.

Stack overflow during startup Переполнение стека во время запуска программы.

Stack override illegal when uninitialized far data present Недопустимое переопределение стека из-за неинициализированных данных в дальнем сегменте.

Unable to grow stack for stack override Невозможно увеличить объем стека для его переопределения.

Затем нужно исключить возможность переполнения диска и аппаратных ошибок при обмене с диском. В этом случае за сообщением о причине прекращения компоновки следует предположение о причине ошибки.

Can't Create Disk Listing File Невозможно создать файл листинга на диске.

Can't Create Debug File Невозможно создать отладочный файл.

Can't Create Listing File Невозможно создать листинг.

Can't Create Output File Невозможно создать выходной файл.

Can't Create Temporary Object File Невозможно создать временный объектный файл.

Can't Open File Невозможно открыть файл (имя).

Can't Open Temporary Object File Невозможно открыть временный объектный файл.

Disk May Be Full Может быть нет места на диске.

Error Opening Temporary File Ошибка открытия временного файла.

Error Reading External Symbols in Ошибка чтения внешних имен в (имя).

Error Reading Global Symbols in Ошибка чтения глобальных имен в (имя).

Error Reading Library Ошибка чтения библиотечного файла.

Error Writing Debug Information Ошибка записи отладочного файла.

Error Writing Disk Listing File Ошибка записи листинга на диск.

Error Writing Debug File Ошибка записи отладочного файла на диск.

Error Writing Linker Output File Ошибка записи выходного файла на диск.

Error Writing Output File. Bad Load Map Data Ошибка записи выходного файла. Плохие данные карты загрузки.

Но все это не вина программиста, а его беда, с которой надо разобраться в смысле состояния аппаратуры и системного математического обеспечения компьютера. Все эти сведения приведены для того, чтобы не искать огрехов в своих программах или действиях.

Перейдем теперь к предупреждениям, порождаемым ошибками программирования или работы с компоновщиком. Начнем с того, что

программист может задать имя какого-либо из файлов с ошибкой. Тогда компоновщик может выдать одно из следующих предупреждений:

Can't Find Assembler Listing File Не обнаружен листинг трансляции.
Can't Find File Не обнаружен входной файл (имя).
Can't Find Linker Data File Не обнаружен файл компоновки (имя).
Input Filename Exceeds Maximum Number of Characters Слишком длинное имя входного файла.
Output Filename Exceeds Maximum Number of Characters Слишком длинное имя выходного файла.

Но в этих случаях нужно просто ввести правильное имя запрашиваемого файла, и компоновка продолжается.

При отсутствии ошибки в имени файла компоновщик читает файл и проверяет правильность его формата. При неправильных форматах объектных файлов выдаются такие предупреждения:

Bad Object File Плохой объектный файл (имя).
Illegal Object File Недопустимый объектный файл (имя).
'FILENAME' is a Packed Object File which can only be used with the Librarian Упакованный объектный файл (имя) может быть использован только через библиотекаря.
'FILENAME' is in the Microsoft Relocation Format Объектный файл (имя) в перемещаемом формате Майкрософт.

В последнем случае даже выдается совет

Use the Microsoft Linker or Put the '.OUTPUT 2500AD' Directive in the Source File Используйте компоновщик Майкрософт или включите директиву '.OUTPUT 2500AD' в исходный текст.

Может ли транслятор знаменитой фирмы работать с системой команд i8051 и следовать этой директиве, автору неизвестно. В случае соответствия формата входного файла компоновщик запрашивает начальные адреса секций, а при явной ошибке выдает предупреждение

Illegal Address Недопустимый адрес.

При выборе опции компоновщика также может случиться ошибка, о чем компоновщик выдает предупреждение

Unknown Linker Option Неизвестная опция компоновщика (буква).

Все эти погрешности легко исправить по ходу работы, так как в этих случаях не требуются переделка исходных текстов и повторная трансляция.

Компоновка прекращается, если превышены возможности транслятора, о чем он выдает следующие сообщения:

Maximum File Count Exceeded Превышено допустимое количество файлов

Maximum Number of Different Sections Exceeded Превышено допустимое количество различных секций.

Maximum Number of Input Files Exceeded Превышено допустимое количество входных файлов.

Но вряд ли кто-нибудь будет компоновать программы из такого большого количества файлов да еще с использованием громадного количества секций. В крайнем случае следует объединить близкие по назначению модули или секции и повторить трансляцию для измененных исходных текстов.

В очень редких случаях могут возникнуть неприятности из-за сбоев в чтении объектных файлов. В этом случае транслятор выдает следующие сообщения:

Unknown External Record Type Не опознается запись типа внешних ссылок (имя).

Unknown object code record type Не опознается запись типа программной в объектном файле (имя).

Unknown Relocation Record Type Не опознается запись типа перемещаемой (имя).

Исправить такие ошибки можно только повторной трансляцией исходного текста для получения качественных записей в объектных файлах. Прекращается компоновка и в случае явной путаницы с именами и адресами, о чем выдаются сообщения:

Descending Addresses Detected in File В файле (имя) обнаружены уменьшающиеся адреса.

Multiple Defined Global Symbol Found in Files Обнаружено повторное определение глобального имени.

Overlapping Addresses Перекрытие адресов.

Symbol Table Buffer Section Overflow Переполнение буфера таблица имен секции.

Unresolved External Reference Неразрешимая внешняя ссылка.

Это уже связано с явной ошибкой программиста, выявить которую можно только анализом листингов компонуемых модулей. В этом случае обязательно

должен быть исправлен исходный текст, в котором обнаружена ошибка, и произведена его трансляция.

А далее могут последовать какие-то из сонма предупреждений, связанных с подозрительными несоответствиями фактически вычисленных при компоновке адресов способам адресации, использованным в командах. Такие несоответствия могут проявиться и при коррекции адресов в случае перемещения модуля на новое место и в случае вычисления адреса для внешней ссылки. В первом случае после текста предупреждения указываются имя файла и номер строки:

in File: (Имя файла) Line Number: (Номер строки)

а во втором выдаются имя глобальной метки и имя файла:

Global Symbol: (Имя глобальной метки) Filename: (Имя файла)

В связи с разнообразием способов адресации некоторая часть предупреждений вряд ли может выдаваться компоновщиком при работе с программами, предназначенными для микроконтроллеров семейства i8051. Но автор скопировал из машинного текста программы все предупреждения и приводит их вместе с переводами "на всякий пожарный случай".

Addressable Bit Relocated Above 127 Адресуемый бит перемещен выше 127.

Call/Jump Relocated Out of Bounds Безусловный переход перемещен за пределы адресации.

External Addressable Bit Relocated Above 127 Внешний адресуемый бит перемещен выше 127.

External Call/Jump Relocated Out of Bounds Внешний безусловный переход перемещен за пределы адресации.

External Data Value Larger Than 8 Bits Значение внешних данных выходит за пределы байта.

External Data Value Relocated Out of Direct Page Внешние данные перенесены за пределы страницы с прямой адресацией.

External Data Value Relocated Out of High 32k Внешние данные перемещены за пределы верхних 32к.

External Data Value Relocated Out of Low 32k Внешние данные перемещены за пределы нижних 32к.

External Data Value Relocated Out of Low 64k Внешние данные перемещены за пределы нижних 64к.

External Data Value Relocated Out of Page 0 Внешние данные перемещены за пределы 0 страницы.

External Jump Relocated Out of Bounds Внешний безусловный переход перемещен за пределы адресации.
Jump Relocated Out of Bounds Безусловный переход перемещен за пределы адресации.
Low Byte of Relocated Segment Must = 0 Младший байт перемещаемого сегмента должен быть равен 0.
Low 8 Bits of External Data Value Segment Not = 0 Младший байт сегмента внешних данных не равен 0.
Operand Relocated Out of 256 Byte Range Операнд перемещен за пределы однобайтового адреса.
Operand Relocated Out of High 32k Операнд перемещен за пределы верхних 32к.
Operand Relocated Out of Low 32k Операнд перемещен за пределы нижних 32к.
Operand Relocated Out of Low 64k Операнд перемещен за пределы нижних 64к.
Operand Relocated Out of Page Операнд перемещен за пределы страницы.
Relative Call/Jump To a Global Symbol In a Different Segment Относительный переход на глобальное имя в другом сегменте.
Relative Operand Too Far Away Относительный операнд расположен слишком далеко.
Relative Reference To a Global Symbol Too Far Away Относительная ссылка на слишком далекое глобальное имя.

Как видите, здесь есть такие предупреждения, которые не относятся к способам адресации i8051. Если после какого-либо из таких предупреждений и будет создан исполняемый файл, то не следует торопиться с загрузкой полученной программы в микроконтроллерное устройство. Следует сначала тщательно разобраться в причине, породившей предупреждение, а еще лучше доработать исходный текст или расположение модулей таким образом, чтобы исключить предупреждения.

Если программист остановил работу компоновщика, то выдается сообщение

Assembly Terminated By Operator Сборка прекращена оператором.

Компоновщик может также выдавать информацию о перемещении файлов в процессе обработки листингов и создания таблиц имен для отладки, но это уже не связано с синтаксическими ошибками в исполняемой программе.

3.9. Форматы некоторых файлов

При переходе от исходного текста к исполняемой программе транслятор и компоновщик осуществляют преобразование исходной информации в машинный код. Ради расширения кругозора любопытно взглянуть, каким образом кодируется эта информация в некоторых типах файлов. В качестве примера ниже показано содержимое файлов для приведенного ранее модуля вычисления НОД. Исходный текст записывается в текстовом формате. А каким образом он кодируется на машинном носителе? Следующая "распечатка" получена при помощи системной программы DEBUG. В этом виде текст занимает 80 байт. Жирным шрифтом выделены шестнадцатеричные коды отображаемых символов и пробелов. Не выделены пары символов, обеспечивающие возврат каретки и перевод строки. Системная программа приводит в правой части распечатки отображаемые символы, а вместо неотображаемых символов выводит точки.

```
65 75 63 6C 69 64 3A 20-44 49 56 20 20 20 20 20   euclid:DIV
41 42 0D 0A 20 20 20 20-20 20 20 20 58 43 48 20   AB..     XCH
20 20 20 20 41 2C 20 42-0D 0A 20 20 20 20 20 20       A, B..
20 20 4A 4E 5A 20 20 20-20 20 65 75 63 6C 69 64     JNZ   euclid
0D 0A 20 20 20 20 20 20-20 20 2E 45 4E 44 0D 0A   ..      .END..
```

Файл с листингом также является текстовым и закодирован аналогичным образом, поэтому нет необходимости приводить его. Объектный файл содержит смешанную информацию. Ниже приведен результат вывода этого файла той же системной программой. По информации в правой части распечатки удается прочесть только имена стандартных сегментов (выделены жирным шрифтом), а пять байтов машинных кодов удалось найти только в самом конце файла (выделены подчеркиванием).

```
5A 0B 00 04 35 02 00 00-3D 01 00 00 00 00 00 00   Z...5...=.......
02 10 FF 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00   ...............
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00   ...............
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00   ...............
00 00 00 00 00 00 00 00-00 00 00 00 00 04 43 4F   ..............CO
44 45 00 00 00 19 54 55-8B EC 53 56 57 8B F1 E8   DE....TU..SVW...
BB FF FF FF 6A 01 8B D8-85 DB 5F 0F 84 2E 00 00   ....j...._.....
00 00 00 00 00 00 00 00-00 00 05 00 00 00 00 00   ...............
00 00 00 00 01 44 41 54-41 00 FF 75 18 8B 06 FF   .....DATA..u....
75 14 FF 75 10 FF 75 0C-FF 75 08 56 FF 50 10 5F   u..u..u..u.V.P._
5E 5B 5D C2 18 00 00 00-00 00 00 00 00 00 00 00   ^[]............
00 00 00 00 00 00 00 00-00 00 00 00 52 53 45 43   ............RSEC
54 00 81 FA 36 4B 80 30-74 2F 8B 44 24 04 89 81   T...6K.0t/.D$...
60 04 00 00 89 91 58 04-00 00 8D 81 00 00 00 00   `.....X.........
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ...............
00 00 00 42 53 45 43 54-00 F3 A5 5E 5F EB E8 55   ...BSECT...^_..U
8B EC 53 56 57 8B 7D 0C-85 FF 0F 84 69 4F 0B 00   ..SVW.}.....iO..
8B 5D 10 00 00 00 00 00-00 00 00 00 00 00 00 00   .]..............
00 00 00 00 00 00 00 00-00 00 04 00 05 00 00 00   ...............
00 00 00 00 00 00 00 00-00 01 01 84 01 02 C5 F0   ...............
01 02 70 FB 7E 00 00 7F                           ..p.~...
```

Как видно из приведенного примера, программа расположена в самом конце объектного файла, а начало его предназначено для хранения разного рода служебной информации, которая используется при редактировании связей. Даже к машинным кодам приставлены какие-то добавочные коды такого же назначения. Их можно рассмотреть на следующем фрагменте

```
01 01 84
01 02 C5 F0
01 02 70 FB
7E
00 00
7F
```

Тому, кто разрабатывает прикладные программы для микроконтроллеров, разбираться в формате объектного файла не нужно. Изучение документации такого рода дело разработчиков транслятора и компоновщика.

Рассмотрим теперь два типа исполняемых файлов, создаваемых компоновщиком. Файл EUCLID.TSK занимает всего пять байтов, которые выведены в шестнадцатеричной записи при помощи программы DEBUG:

```
84 C5 F0 70 FB
```

Это наиболее компактная форма выдачи файла, но ее недостатком является невозможность его проверки на отсутствие ошибок записи. Содержимое файла EUCLID.HEX можно прочитать при помощи редактора, так как в этом файле использован текстовый формат представления шестнадцатеричных данных. Кроме того, в состав файла включены контрольные суммы для подтверждения корректности записей:

```
:0500000084C5F070FB57
:00000001FF
```

Каждая строка, начинающаяся с символа двоеточия и заканчивающаяся парой символов — возврат каретки и перевод строки, представляет собой запись из пяти полей. Первое поле занимает один байт и содержит число, задающее количество байтов данных в записи. Второе поле занимает два байта и задает адрес загрузки первого байта данных. Третье поле занимает один байт и содержит код записи (00 для данных и 01 для последней записи файла). Далее следует поле данных длиной от 0 до 255 байтов. Последнее поле содержит один байт контрольной суммы. Младший байт суммы всех байтов записи должен быть равен 0.

В соответствии с приведенными сведениями о полях записей проведем разбор нашего примера

```
:                 символ начала записи
05                количество символов в 1-й записи
0000              загрузочный адрес
00                признак данных
84 C5 F0 70 FB    загружаемые данные (5 байт)
57                контрольная сумма записи
:                 символ начала записи
00                количество символов во 2-й записи
0000              загрузочный адрес
01                признак конца файла (далее 0 байт данных)
FF                контрольная сумма записи
```

Для первой записи сумма всех байтов равна 400, а для второй — 100. Результаты проверки свидетельствуют о корректности обеих записей.

Файл с расширением MAP имеет текстовый формат, так что прочесть его содержимое при помощи текстового редактора не представляет труда. Приведем пример файла карты загрузки:

```
Global Symbol Name          Global Value       Global Filename
**************************************************************
*                     L O A D     M A P                       *
**************************************************************
*  Section Name  Starting Address  Ending Address  Size      *
**************************************************************
*  euclid.obj                                                 *
*  CODE          0000              0004            0005      *
**************************************************************
             Linker Output Filename : euclid.hex
             Disk Listing  Filename : euclid.map
             Symbol Table  Filename : <None Specified>
       Link Errors :  0            Output Format :  Intel Hex
```

Форматы файлов таблицы имен гораздо более разнообразны, что, по-видимому, связано с различием отладчиков, используемых разными фирмами. Файлы этого типа имеют расширение SYM и содержат имена и значения различных объектов программ. Общим у всех этих файлов является то, что каждой паре имя-значение отводится отдельная запись. А различия связаны с расположением полей и со способами кодирования этих записей. Записи в таблицах глобальных имен фирмы 2500 A.D., получаемые для опций S и A, содержат длинные и короткие имена соответственно. Первое поле записи отводится для имени и занимает 32 байт или 10 байт соответственно. Следующие два байта отводятся на адрес (старший байт предшествует младшему). Третье поле размером один байт отводится на номер файла. Последнее поле (36-й или 14-й байт соответственно) размером один байт зарезервировано. Признаком конца файла в обоих случаях служит байт со значением 0FFh. Файлы с длинными именами отличаются байтом со значением 0E0h, расположенным в начале файла.

Глава 4

Программирование арифметических действий

4.1. Кодирование информации в микроконтроллере

Микроконтроллер и управляемый им объект входят в состав изделия, взаимодействующего с пользователем. При помощи периферийных устройств микроконтроллер получает информацию о состоянии объекта и выдает команды управления объектом, а также воспринимает команды пользователя и выдает сигналы пользователю. Вся информация для осуществления этих действий представляется в микроконтроллере в виде двоичных кодов. На начальном этапе программирования необходимо четко определить полный состав информации, которая используется для взаимодействия микроконтроллера с управляемым объектом и с пользователем изделия, и выбрать рациональные способы кодирования этой информации. Программисту приходится заниматься всеми подробностями кодирования информации в связи с отсутствием операционной системы и сервисных программ в условиях строго ограниченных ресурсов.

Кодирование информации для обмена с периферией определяется интерфейсом микросхем, с которыми связан микроконтроллер, и/или электрической схемой устройства. При выборе способа кодирования информации, используемой в микроконтроллере для решения задач управления объектом, программист должен исходить из критериев наиболее эффективного ее представления с точки зрения использования объема памяти и скорости работы программы. Поэтому кодирование внутренних

переменных в микроконтроллере должно соответствовать системе команд, используемых при их обработке. Для обработки внутренних переменных, которые могут быть представлены неотрицательными целыми числами в формате байта, наиболее целесообразно использовать 8-разрядный позиционный двоичный код. В этом случае система команд микроконтроллеров семейства i8051 обеспечивает обработку информации посредством прямого и обратного счета, сложения, вычитания, умножения и деления.

Но в микроконтроллере приходится кодировать значительно более разнообразную числовую информацию. На практике число может быть получено одним из двух принципиально различных способов: счетом или измерением. В случае счета количества предметов или определения их порядковых номеров получаются целые числа. Неправильное (не соответствующее истинному положению вещей) определение целого числа является *ошибкой* (mistake). В случае измерения числа получаются сопоставлением двух сопоставимых характеристик объектов или процессов, одна из которых может изменяться, а другая выбрана в качестве эталона. При этом число не обязательно должно быть целым и в принципе определяется с некоторой *погрешностью* (error). При выборе способа представления чисел нужно четко различать, какого рода числа используются и обрабатываются программой. Использование двоичных кодов в микроконтроллере не исключает возможностей представления чисел в десятичной системе счисления. Но ее лучше применять для взаимодействия с пользователем, а для внутреннего представления чисел целесообразнее использовать двоичную систему.

Начнем с кодирования положительных целых чисел в позиционной системе, характеризующейся использованием символов, называемых цифрами. Общее количество цифр, используемых в любой системе счисления, равно ее основанию (включая цифру 0). Цифра 0 представляет никакое количество независимо от положения в записи числа. Количественное значение остальных цифр зависит от их позиции (положения) в записи числа. Поясним сказанное на примере двоичной системы счисления, использующей наименьшее количество цифр: 0 и 1. Количественное значение цифры 1 рассмотрим на примере представления положительного целого числа байтом, состоящим из 8 двоичных разрядов с номерами от нулевого (самый младший разряд расположен справа) до седьмого (самый старший разряд расположен слева). "Вес" цифры 1 для 0, 1, 2, 3, 4, 5, 6 и 7 разрядов не одинаков и составляет соответственно 1, 2, 4, 8, 16, 32, 64 и 128, то есть удваивается при переходе на 1 разряд влево. Таким образом 1 байт может быть использован для представления всех (без изъятия!)

положительных целых чисел от 0 до 255. При необходимости записи положительных чисел превышающих 255 нужно использовать более 1 байт. Например, двоичное "слово", состоящее из двух байтов, может представлять все положительные целые числа от 0 до 65535.

Но это не единственный способ кодирования положительных целых чисел. В технике часто используется двоичный рефлексный код, называемый также кодом Грея. Ниже приведена таблица 4-разрядных позиционного и рефлексного двоичных кодов.

Десятичное число	Шестнадцатеричная цифра	Позиционный код	Рефлексный код
0	0	0000	0000
1	1	0001	0001
2	2	0010	0011
3	3	0011	0010
4	4	0100	0110
5	5	0101	0111
6	6	0110	0101
7	7	0111	0100
8	8	1000	1100
9	9	1001	1101
10	A	1010	1111
11	B	1011	1110
12	C	1100	1010
13	D	1101	1011
14	E	1110	1001
15	F	1111	1000

Рефлексный код удобен для использования в некоторых типах преобразователей аналогового сигнала в цифровой. Изменение содержимого только одного разряда при переходе сигнала с одного уровня на другой обеспечивает отсутствие ошибок преобразования, хотя оно не может устранить погрешность. Для тех, кого интересует общее правило получения рефлексного кода, укажем алгоритм его получения из позиционного. Для получения рефлексного кода нужно вычислить функцию неравнозначности (ИСКЛЮЧАЮЩЕЕ ИЛИ) от исходного числа и его частного, полученного делением на два. На практике бывает необходимо выполнять обратное преобразование, но оно немного сложнее и будет рассмотрено позднее.

Преимущество позиционного представления чисел состоит в том, что, во-первых, для представления любых сколь угодно больших чисел используются одни и те же цифры, во-вторых, все арифметические операции со сколь угодно большими числами могут быть выполнены как конечная последовательность единообразных действий с цифрами, представленными в отдельных разрядах этих чисел. Разумеется, при этом

необходимо выполнять эти действия в определенной последовательности и учитывать результат выполнения действий с предшествующими разрядами. А правила всех поразрядных арифметических операций сводятся к таблице сложения и таблице умножения, имеющих для двоичной системы счисления наипростейший (по сравнению с остальными системами счисления) вид:

```
0 + 0 =  0          0 * 0 = 0
0 + 1 =  1          0 * 1 = 0
1 + 0 =  1          1 * 0 = 0
1 + 1 = 10          1 * 1 = 1
```

Таблица умножения в двоичной системе оказалась проще таблицы сложения потому, что произведение двух цифр не выходит за пределы одного разряда, а значение суммы может выйти за его пределы. Это всем известный перенос, который школьники учатся "держать в уме". По этой причине таблица сложения должна быть дополнена для случая переноса из предыдущего разряда.

```
        Перенос 0 Перенос 1
0 + 0 =         0         1
0 + 1 =         1        10
1 + 0 =         1        10
1 + 1 =        10        11
```

Приведенных сведений вкупе с опытом школьных упражнений по сложению, вычитанию, умножению и делению "столбиком" (как говаривали старые учителя, "по Маленину-Буренину") в последующем будет вполне достаточно для самостоятельной проверки читателями корректности приведенных программ арифметических вычислений.

При прямом счете, сложении и умножении положительных чисел всегда получаются положительные числа. В микроконтроллерах семейства i8051 выход результата прямого счета за пределы разрядной сетки никак не контролируется. Выход результата сложения положительных чисел за пределы одного байта приводит к установке бита переноса в 1. Выход результата умножения положительных чисел за пределы одного байта приводит к установке бита переполнения в 1. Если результат операции выходит за пределы байта, то для его правильного представления нужно использовать дополнительные двоичные разряды. Способ кодирования при этом не изменяется, но нужно рассмотреть способы обработки чисел, представленных несколькими байтами. Результатом обратного счета или вычитания могут быть отрицательные числа. Получение отрицательного числа в результате обратного счета никак не контролируется. Выход результата вычитания за пределы представления положительного числа

приводит к установке бита переноса в 1. Для работы с отрицательными числами необходимо рассмотреть, во-первых, вопросы их кодирования, а во-вторых, возможности обработки этих кодов существующим набором команд. Отложив программирование арифметических действий с многобайтными и отрицательными кодами, остановимся на кодировании отрицательных чисел.

Поскольку одним байтом может быть закодировано только 255 чисел, приходится ограничиться представлением положительных чисел до +127, при этом 7 разряд используется для кодирования знака и для положительных чисел принимается равным 0. Для отрицательных чисел можно принять значение 7 разряда равным 1. Остается выбрать кодирование для остальных разрядов. Традиционно применяемый в текстах способ записи числа в виде знака и модуля использовался и для двоичного кодирования в некоторых вычислительных машинах, потому что был удобным для аппаратной реализации умножения и деления. Для системы команд i8051 при кодировании чисел знаком и модулем ни одна арифметическая операция не обеспечит корректных результатов. Однако существует кодирование, при котором корректно выполняются операции прямого и обратного счета, сложение и вычитание. Приведем пример обратного счета в случае 4-разрядного двоичного кода от числа +7 до числа –8.

Десятичное число	Двоичный код
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Для отрицательных чисел такой код называется *дополнительным,* потому что он дополняет код соответствующего (равного по модулю) положительного числа до количества кодируемых чисел (для 4 разрядов — до 16, для байта — до 256, а для слова — до 65536). (Для числа –8 в этой таблице

соответствующего положительного числа нет, а число 0 неотрицательное.) При кодировании отрицательных чисел дополнительным кодом знаковому разряду байта нужно приписать вес –128, а знаковому разряду слова — вес –32768. Такой вариант кодирования позволяет представить одним байтом все числа от –128 до +127., а двумя байтами — числа от –32768 до +32767.

Арифметические операции ADD (ADDC) и SUBB при использовании дополнительного кода выполняются корректно. Если результат сложения или вычитания выходит за пределы представления чисел со знаком, то бит переполнения устанавливается в 1. Следует отметить нежелательность использования в однобайтовых арифметических операциях кодов, представляющих –128 для одного байта и –32768 для двух байтов, из-за отсутствия кодов представляющих соответствующие положительные числа. В операции обратного счета DEC результатом уменьшения 0 на 1 является код, соответствующий числу 255 для кодирования чисел без знака или числу –1 для кодирования со знаком. При выполнении этой операции признак переполнения также не вырабатывается. Аналогичным образом работает и INC, так что результаты обеих операций будут неверны при выходе за пределы представления чисел заданным кодом. При кодировании целых чисел без знака это будет увеличение 255 для INC и уменьшение 0 для DEC, а при кодировании целых чисел со знаком — увеличение 127 и уменьшение –128 соответственно. Но выполнение этих команд не влияет на содержимое битов переноса и переполнения. Арифметические операции MUL и DIV не обеспечивают корректных результатов и при кодировании отрицательных чисел дополнительным кодом, поэтому умножение и деление с отрицательными операндами нужно программировать особо.

До тех пор пока арифметические операции (кроме деления) используются для работы с целыми числами, полученными посредством счета, их результаты будут точными. Для представления результата операции деления в общем случае нужны дробные числа. Дробные числа нужны и для представления чисел, полученных посредством измерений. В этом случае есть два подхода. Наиболее универсальным является представление чисел при помощи порядка и мантиссы. Но в микроконтроллере использование чисел с плавающей запятой вряд ли целесообразно с точки зрения расхода его ресурсов, хотя для карманных калькуляторов это оправдано их назначением. Более рационально применять такой выбор масштабов, чтобы использовать целые числа или числа с фиксированной запятой. Поэтому здесь рассматривается только программирование арифметических операций с целыми числами.

4.2. Арифметические действия с большими числами

В микроконтроллерах семейства i8051 отсутствует набор команд для осуществления арифметических действий в тех случаях, когда для представления чисел приходится использовать более одного байта. Поэтому рассмотрим, каким образом можно осуществлять 4 действия арифметики для положительных чисел, представляемых двумя байтами. Для программирования арифметических действий с такими числами следует иметь в виду, что вес старшего байта в 256 раз больше, чем у его соседа справа. Обозначая младший байт индексом 0, а старший — индексом 1, представим операнды выражениями 256*X(1) + X(0) и 256*Y(1) + Y(0). Здесь X(0) и Y(0) — числовые значения младших байтов операндов, а X(1) и Y(1) — числовые значения старших байтов операндов. Представленные выражения можно складывать, вычитать и умножать, но не делить.

В результате сложения получим следующее выражение для суммы:

```
256*(X(1) + Y(1)) + (X(0) + Y(0))
```

из которого видно, что для вычисления младшего байта суммы нужно сложить младшие байты операндов, а для вычисления старшего байта суммы — старшие байты операндов. Но независимого сложения старших и младших байтов недостаточно, так как в случае переполнения суммы младших байтов нужно добавить к сумме старших байтов 1. Поэтому нужно сначала вычислить сумму младших байтов командой ADD, а затем — сумму старших байтов с учетом переноса, то есть командой ADDC. Пусть первый операнд записан в регистрах R0 и R1, а второй — в регистрах R2 и R3, причем в регистрах с четными номерами хранятся младшие байты. Составим программу суммирования с записью суммы по адресу первого операнда:

```
MOV     A, R0     ;мл. байт первого числа в накопителе
ADD     A, R2     ;добавлен мл. байт второго числа
MOV     R0, A     ;запоминание мл. байта суммы
MOV     A, R1     ;ст. байт первого числа в накопителе
ADDC    A, R3     ;добавлен ст. байт второго числа и перенос
MOV     R1, A     ;запоминание ст. байта суммы
```

Следует заметить, что при выполнении команд пересылки кодов состояние бита переноса не изменяется. Для вычитания нужно проделать аналогичные действия, но нужно учесть одну тонкость. Дело в том, что в системе команд i8051 отсутствует обычное вычитание, поэтому программа вычитания должна начинаться с очистки бита переноса:

```
CLR     C          ;очистка бита переноса
MOV     A, R0      ;мл. байт первого числа в накопителе
SUBB    A, R2      ;вычитание мл. байта второго числа
MOV     R0, A      ;запоминание мл. байта разности
MOV     A, R1      ;ст. байт первого числа в накопителе
SUBB    A, R3      ;вычитание ст. байта второго числа и займа
MOV     R1, A      ;запоминание ст. байта разности
```

Разница между действиями сложения и вычитания состоит еще в том, что при сложении нужно учитывать перенос, а при вычитании — заём. Но для хранения займа при вычитании используется тот же самый бит переноса. В обоих примерах использована регистровая адресация операндов и результата.

При помощи косвенной адресации можно написать более компактную программу сложения или вычитания чисел, состоящих из любого количества байтов (лишь бы хватило памяти). Пусть младший байт первого операнда записан в ячейке с именем frst, а все последующие — в следующих ячейках. Аналогичным образом байты второго операнда должны быть записаны в массив, первая ячейка которого имеет имя scnd. Предположим, что оба числа состоят из 5 байт. Для резервирования памяти запишем

```
       .RSECT
frst:  .DS     5        ;5 байт ОЗУ для первого операнда
scnd:  .DS     5        ;5 байт ОЗУ для второго операнда
```

Считая, что запись операндов в эти ячейки осуществлена в другой части программы, напишем программу сложения первого числа со вторым и с записью суммы на место первого числа:

```
       .CODE
       MOV    R0, #frst   ;запись адреса 1-го операнда в регистр
       MOV    R1, #scnd   ;запись адреса 2-го операнда в регистр
       MOV    R3, #5      ;запись количества байтов в регистр
       CLR    C           ;очистка бита переноса
addm:  MOV    A, @R0      ;байт первого числа в накопителе
       ADDC   A, @R1      ;добавление байта 2-го операнда
       MOV    @R0, A      ;запоминание байта суммы
       INC    R0          ;вычисление адреса байта 1-го числа
       INC    R1          ;вычисление адреса байта 2-го числа
       DJNZ   R3, addm    ;счет количества необработанных байтов
```

Три команды в самом начале программы используют непосредственную адресацию источника. Притом для двух из них транслятор осуществляет подстановку фактических значений адресов младших байтов операндов. Команда очистки бита переноса нужна потому, что в циклической части программы используется команда сложения с учетом переноса. За счет использования косвенной адресации удается обработать все байты первого

и второго операндов одними и теми же командами. А для того чтобы в следующем цикле обратиться к следующим ячейкам ОЗУ, содержимое регистров R0 и R1 надо увеличивать на 1. Счет количества необработанных байтов производится последней командой. При ее выполнении число в регистре R3 уменьшается на 1, и если результат не равен 0, то управление передается на начало цикла. При показанном ранее способе для сложения 5 пар байтов пришлось бы использовать 15 команд, в приведенном примере их только 10. Кроме экономии памяти программ этот фрагмент более универсален. С другой стороны длительность работы этой программы больше, так как для ее завершения нужно выполнить 34 команды вместо 15. Таким образом, экономия одного ресурса, как правило, осуществляется за счет затраты других.

Переходя к двум другим арифметическим действиям, следует отметить, что умножение или деление двоичного числа, состоящего из нескольких байтов, на целую степень двойки осуществляется при помощи сдвига всех байтов этого числа влево или вправо. Поскольку вес двоичной цифры 1 возрастает вдвое при переходе в соседний левый разряд, то для умножения на 2 нужен сдвиг влево на 1 разряд, для умножения на 4 — на 2 разряда и так далее. Для передачи битов кода числа из одного байта в другой нужно использовать операцию циклического сдвига влево с участием бита переноса. Поскольку перед очередным сдвигом крайнего байта значение бита переноса может быть произвольным, надо или устанавливать нужное значение или после завершения сдвигов корректировать содержимое соответствующего количества разрядов младшего или старшего байта. Умножение на заранее заданную константу, в коде которой содержится небольшое количество единиц, также может производиться последовательностью операций сдвига и суммирования. Однако в том случае, когда значение множителя заранее неизвестно, нужно использовать операции умножения.

При перемножении чисел, каждое из которых представлено двумя байтами, произведение может занять 4 байт. Перенумеруем их от младших к старшим байтам 0, 1, 2 и 3. Для вычисления произведения можно воспользоваться формулой

```
65536*(X(1)*Y(1)) + 256*(X(1)*Y(0) + X(0)*Y(1)) + X(0)*Y(0).
```

Из формулы следует, что результат должен получаться накоплением, причем младший байт произведения младших байтов должен быть записан в 0 байт, а старший — в 1. Произведения младшего байта на старший и старшего байта на младший добавляются соответственно к первому и второму байтам. Произведение старших байтов добавляется ко второму и третьему байтам. Общее правило таково: сдвиг произведения байтов

относительно младшего байта результата равен сумме сдвигов относительно младших байтов множимого и множителя. Для уменьшения количества операций сложения частичных произведений целесообразно начинать с вычисления произведений младших байтов (вспомните умножение "столбиком"!). Пусть сомножители находятся в регистрах R1, R0 и R3, R2. Следующая программа помещает произведение в регистры R7, R6, R5 и R4 (старшие байты находятся в регистрах с большими номерами):

```
      MOV   A, R0
      MOV   B, R2
      MUL   AB       ;произведение мл. байтов
      MOV   R4, A    ;в 0-ой байт произведения
      MOV   R5, B    ;в 1-ый байт произведения
      MOV   A, R1
      MOV   B, R3
      MUL   AB       ;произведение ст. байтов
      MOV   R6, A    ;во 2-ой байт произведения
      MOV   R7, B    ;в 3-ий байт произведения
      MOV   A, R1
      MOV   B, R2
      MUL   AB       ;произведение ст. байта на мл. байт
      ADD   A, R5
      MOV   R5, A    ;в 1-ый байт произведения
      MOV   A, R6
      ADDC  A, B
      JNC   ncy1     ;переход, если нет переноса в 3-й байт
      INC   R7       ;коррекция 3-го байта произведения
ncy1: MOV   R6, A    ;во 2-ой байт произведения
      MOV   A, R0
      MOV   B, R3
      MUL   AB       ;произведение мл. байта на ст. байт
      ADD   A, R5
      MOV   R5, A    ;в 1-ый байт произведения
      MOV   A, R6
      ADDC  A, B
      JNC   ncy2     ;переход, если нет переноса в 3-й байт
      INC   R7       ;коррекция 3-го байта произведения
ncy2: MOV   R6, A    ;во 2-ой байт произведения
```

В приведенном примере после записи произведения младших байтов производится вычисление произведения старших, поскольку их результаты записываются в те байты результата, которые не перекрываются. Затем к ним добавляются произведения старшего на младший и младшего на старший. Для вычисления произведения пришлось использовать 4 команды умножения, 4 команды сложения и много операций пересылки.

Как видно из приведенных примеров, программы для сложения, вычитания и умножения больших чисел не представляют особых трудностей. Нужно только написать расчетные формулы, в соответствии с которыми должен быть составлен исходный текст программы.

Программирование деления больших чисел затрудняется из-за отсутствия конечных формул для вычисления частного и остатка. Рассмотрим подробно в качестве примера задачу вычисления частного от деления двухбайтового числа на однобайтовое. С использованием прежних обозначений запишем выражение для деления первого числа на второе в виде дроби

```
(256*X(1) + X(0)) / Y(0).
```

Операция целочисленного деления заключается в приведении неправильной дроби к правильной, при этом целая часть числа является частным, а числитель дробной части — остатком. Обозначим частное и остаток буквами Q и R соответственно. Тогда задача целочисленного деления сводится к нахождению двух неотрицательных целых чисел, удовлетворяющих уравнению и неравенству

```
x = Q * y + R
R < y
```

Это уравнение решается подбором частного, то есть методом проб и ошибок. Например, это можно делать многократным вычитанием делителя из делимого до тех пор, пока остаток не будет меньше делителя. Тогда частное равно количеству операций вычитания. Более экономным является метод пробных вычислений разности между делимым и произведением делителя на приближенное значение частного. При этом сначала подбирается старшая цифра частного как наибольшее из значений, при котором остаток еще положителен, затем следующие цифры. Это известный всем школьникам (по крайней мере, до внедрения карманных калькуляторов) метод деления "столбиком". Для двоичного кодирования чисел этот метод наиболее эффективен, так как позволяет при каждой пробе вычислять очередную цифру частного. Пусть делимое находится в регистрах R3, R2, а делитель — в регистре R0. Отведем для запоминания текущего остатка регистр R4, а для счета количества цифр частного — регистр R1. При делении "столбиком" произведение делителя на очередную цифру частного сдвигается вправо на 1 разряд относительно делимого. При программировании целесообразнее сдвигать остаток влево. Тогда в освобождающиеся биты регистров, используемых для хранения делимого, можно записывать значения очередных битов частного. После завершения программы частное будет записано на месте делимого.

```
        MOV     B, R0           ;запись делителя в регистр B
        MOV     R1, #16         ;количество разрядов делимого
        MOV     R4, #0          ;заготовка для остатка
dwb3:   CLR     C               ;очистка очередного бита для
                                ;частного
        MOV     A, R2           ;
        RLC     A               ;сдвиг мл. разрядов частного
        MOV     R2, A           ;
        MOV     A, R3           ;
        RLC     A               ;сдвиг ст. разрядов частного
        MOV     R3, A           ;
        MOV     A, R4           ;
        RLC     A               ;сдвиг текущего остатка
        CJNE    A, B, dwb1      ;сравнение текущего остатка с
                                ;делителем
dwb1:   JC      dwb2            ;переход, если остаток меньше
                                ;делителя
        SUBB    A, B            ;вычитание делителя из текущего
                                ;остатка
        INC     R2              ;запись 1 в очередной разряд
                                ;частного
dwb2:   MOV     R4, A           ;
        DJNZ    R1, dwb3        ;16-кратное повторение цикла
```

Использование операции сравнения чисел позволяет обойтись без пробного вычитания делителя из текущего остатка. Если вычитание возможно, то в очередную цифру частного заносится 1. Хотя эта программа занимает немного места в памяти, она выполняется гораздо дольше предыдущих примеров, так как входящие в цикл команды будут выполнены 16 раз. Следует заметить, что в случае делителя, состоящего из нескольких байтов, целесообразнее сначала сравнивать старшие байты остатка и делителя, а в случае их равенства переходить на сравнение младших.

Программа для деления двухбайтового числа на однобайтовое может быть усовершенствована за счет получения старшего байта частного командой деления:

```
        MOV     A, R3           ;ст. байт делимого
        MOV     B, R0           ;запись делителя в регистр B
        DIV     AB              ;
        MOV     R3, A           ;ст. байт частного
        MOV     A, B            ;текущий остаток
        MOV     B, R0           ;запись делителя в регистр B
        MOV     R1, #8          ;количество разрядов остатка
dwb3:   CLR     C               ;очистка очередного бита для
                                ;частного
```

```
        XCH    A, R2          ;
        RLC    A              ;сдвиг мл. разрядов частного
        XCH    A, R2          ;
        RLC    A              ;сдвиг текущего остатка
        CJNE   A, B, dwb1     ;сравнение текущего остатка с
                              ;делителем
dwb1:   JC     dwb2           ;переход, если остаток меньше
                              ;делителя
        SUBB   A, B           ;вычитание делителя из
                              ;текущего остатка
        INC    R2             ;запись 1 в очередной разряд
                              ;частного
dwb2:   DJNZ   R1, dwb3       ;8-кратное повторение цикла
```

В отличие от предыдущего варианта регистр R4 здесь не используется, и остаток записывается в накопитель. По объему эти программы практически одинаковы. Но вторая программа работает намного быстрее первой, поскольку удалось уменьшить как количество циклов, так и количество команд в цикле. В случае делителя, который представлен несколькими байтами, нужно использовать алгоритм первого варианта.

Завершая раздел о принципах программирования для чисел, представляемых несколькими байтами, следует заметить, что нужно избегать их использования без особой необходимости. Но если обойтись без больших чисел невозможно, то при программировании нужно принимать во внимание не только формулы и алгоритмы вычислений, но и ограничения по ресурсам. В зависимости от поставленной задачи может потребоваться оптимизация программы по объему ОЗУ, по объему ПЗУ или по быстродействию. Написать программу, одновременно минимизирующую потребление всех этих ресурсов, невозможно.

4.3. Арифметические действия с отрицательными числами

Команды умножения и деления дают правильные результаты только в случае положительных операндов. Поэтому для правильного вычисления произведения или частного в случае отрицательных операндов требуется сначала вычислить модули этих операндов. После умножения или деления модулей в случае необходимости можно вернуться к первоначальному кодированию с учетом знака результата. Таким образом, для умножения и деления чисел, хотя бы одно из которых отрицательное, нужно знать способы вычисления модуля числа и вычисления знака результата.

Знак результата умножения или деления двух чисел будет отрицательным тогда, когда только одно из чисел отрицательное. Пусть знак первого числа записан в старшем бите регистра R0, а знак второго числа — в старшем бите регистра R1. Тогда знак результата можно записать в бит F0 при помощи четырех команд:

```
MOV   A, R0          ;знака в старший бит накопителя
XRL   A, R1          ;вычисление знака результата
RLC   A              ;старший бит накопителя в флаге
                     ;переноса
MOV   F0, C          ;знака результата в бите F0
```

Указанный бит удобен для запоминания знака результата потому, что на него не влияют результаты команд, выполняющих арифметические операции.

В i8051 отсутствует операция изменения знака числа, однако она может быть выполнена вычитанием этого числа из 0. Предположим, что нам нужно изменить знак однобайтового числа, записанного в регистр R0. Для этого нужно очистить бит переноса и накопитель, вычесть из него содержимое регистра и переслать результат из накопителя в регистр:

```
CLR   C              ;очистка бита переноса
CLR   A              ;очистка накопителя
SUBB  A, R0          ;вычитание
MOV   R0, A          ;запоминание результата
```

Однако имеется другой алгоритм изменения знака числа, использующий следующую закономерность кодирования отрицательных чисел:

```
+0   00000000        -0   00000000 = 11111111+1
+1   00000001        -1   11111111 = 11111110+1
+2   00000010        -2   11111110 = 11111101+1
...  ...             ...  ...
```

Из приведенного примера видно, что нужно сделать для изменения знака числа на противоположный. Сначала необходимо изменить содержимое всех разрядов исходного числа на обратные значения (*обратный* код), а затем добавить единицу в младший разряд числа (*дополнительный* код).

Использование этого алгоритма для изменения знака однобайтового числа не дает никакого выигрыша, если подлежащее преобразованию число записано в регистр. Но для изменения знака числа, записанного в накопитель, гораздо проще воспользоваться следующей парой команд.

```
CPL   A              ;вычисление обратного кода
INC   A              ;добавление 1
```

Для изменения знака двухбайтового числа необходимо сначала получить обратные коды младшего и старшего байтов, а затем добавить 1 к

младшему байту. Пусть младший байт числа записан в накопителе, а старший — в регистре B. Число с обратным знаком получается в том же месте, что и исходное.

```
        CPL     A                   ;вычисление обратного кода мл.
                                    ;байта
        XRL     B, #FFh             ;вычисление обратного кода ст.
                                    ;байта
        ADD     A, #01h             ;добавление 1
        JNC     nc                  ;переход по отсутствию переноса
        INC     B                   ;коррекция ст. байта
nc:     NOP                         ;для записи метки
```

Обратите внимание на то, как в случае переноса из младшего байта старший байт увеличивается на 1. Для получения переноса из младшего байта нужно использовать в программе не INC, а ADD. Код операции NOP записан для того, чтобы не оставлять пустой строку с меткой. В реальной программе в этой строке должна быть записана первая команда следующего блока.

В качестве примера приведем программу перемножения однобайтовых чисел с произвольным знаком. Исходные числа находятся в регистрах R0, R1, а произведение получается в регистре B (старший байт) и накопителе (младший байт).

```
        MOV     A, R0               ;множимое
        MOV     C, A.7              ;знак первого числа в бите C
        JNC     mls1                ;переход по положительному
                                    ;множимому
        CPL     A                   ;вычисление обратного кода
                                    ;множимого
        INC     A                   ;вычисление модуля множимого
mls1:   MOV     B, R1               ;множитель
        JNB     B,7 mls2            ;переход по положительному
                                    ;множимому
        CPL     C                   ;вычисление знака произведения
        XRL     B, #FFh             ;вычисление обратного кода
                                    ;множителя
        INC     B                   ;вычисление модуля множителя
mls2:   MOV     F0, C               ;запоминание знака произведения
        MUL     AB                  ;вычисление произведения
                                    ;модулей
        JNB     F0, mls3            ;переход по положительному
                                    ;произведению
        CPL     A                   ;вычисление обратного кода мл.
                                    ;байта
        XRL     B, #FFh             ;вычисление обратного кода ст.
```

```
                                ;байта
        ADD     A, #01h         ;добавление 1
        JNC     mls3            ;переход по отсутствию переноса
        INC     B               ;коррекция ст. байта
mls3:   NOP                     ;для записи метки
```

Смысл производимых в этой программе действий должен быть понятен по ранее приведенным примерам для вычисления знака произведения и изменения знака числа. Запоминание знака числа необходимо потому, что операция умножения записывает 0 в бит переноса. При умножении и делении чисел, представленных несколькими байтами, изменение знака для вычисления модуля исходных чисел и кода результата требуют выполнения большего количества команд.

Умножение отрицательного числа на целую степень двойки при помощи сдвига ни чем не отличается от умножения положительного числа. При делении отрицательного числа нужно при каждом сдвиге записывать в старший бит старшего байта 1. Для этого после записи старшего байта числа в накопитель перед командой сдвига влево нужно скопировать седьмой бит накопителя в бит переноса.

4.4. Контроль точности при программировании арифметических операций

Читателям может показаться странным этот заголовок: ведь каждая из арифметических операций выполняется микроконтроллером с абсолютной точностью. Но абсолютная точность получается только в случае программирования вычислений с целыми числами ограниченной величины. В общем случае при неправильном программировании результат вычислений может иметь недопустимо большую погрешность или быть ошибочным. Первый случай относится к потере значимости, когда в отведенных для кодирования числа разрядах используются только самые младшие. Второй случай относится к переполнению разрядной сетки, когда число не помещается в отведенных для его кодирования разрядах. Поэтому при программировании вычислений необходимо выбирать такой масштаб представления чисел, чтобы, с одной стороны, полнее использовать разрядную сетку, а с другой — не допустить ее переполнения. Если подбором масштабов не удается решить задачу приведения результатов измерений к целочисленному представлению, то можно использовать представление чисел с фиксированной запятой (в англоязычных странах

для разделения целой и дробной частей числа используется точка). Притом позицию запятой можно фиксировать между разрядами какого-то байта или между байтами, представляющими число. Использование представления чисел с фиксированной запятой не означает какого-либо дополнительного кодирования. Нужно только внести эти сведения в документацию на программы (например, в комментарии) и учитывать при программировании. При использовании представления чисел с фиксированной запятой перед сложением и вычитанием чисел с разными положениями запятой требуется сдвиг одного из операндов, а после умножения и деления почти всегда нужен сдвиг результата.

При обработке измерений может оказаться, что младшие результаты кода числа не достоверны вследствие воздействия шумов и помех. Этот случай потери точности будет рассмотрен в главе о программировании фильтрации сигналов. Во избежание накопления дополнительных погрешностей вычислений целесообразно производить вычисления с результатами измерений, обработанными таким образом, чтобы погрешности измерений были сведены к минимуму.

При использовании байтового формата абсолютная погрешность представления чисел не может быть меньше 0,4% от максимального значения для положительных чисел или 0,8% от максимального модуля для положительных и отрицательных чисел. Если не удается подобрать масштаб и/или требуются меньшие погрешности, то приходится представлять соответствующие числа несколькими байтами. При использовании двух байтов погрешность представления чисел может быть значительно снижена. Результаты вычислений будут содержать большие или меньшие погрешности в зависимости от погрешностей исходных чисел и от действий, произведенных над этими числами. Известно, что при сложении и вычитании абсолютная погрешность результата ограничена суммой абсолютных погрешностей операндов, а при умножении и делении относительная погрешность результата ограничена суммой относительных погрешностей операндов. Если модули операндов близки, то при вычитании для одинаковых знаков операндов или сложении для разных знаков погрешность может превысить результат операции. Поэтому следует тщательно проверять расчетные формулы с точки зрения накопления погрешностей счета. Зачастую две математически равноценные формулы могут давать существенно отличающиеся результаты, так как в дискретной математике переместительный и сочетательный законы выполняются не всегда.

В качестве тривиального примера источника систематической погрешности можно привести операцию целочисленного деления. Поскольку частное в этом случае всегда получается с недостатком, для уменьшения

математического ожидания погрешности вычисления частного нужно перед операцией деления прибавить к делимому половину делителя. Если сумма делимого и половины делителя не превосходит 255, то это можно сделать добавлением трех команд:

```
RLC    A          ;удвоить делимое
ADD    A, B       ;прибавить делитель
RRC    A          ;разделить сумму на 2
DIV    AB         ;вычислить частное
```

Аналогичным образом можно более или менее существенно уменьшить систематические погрешности вычислений, если таковые удается выявить при помощи анализа расчетных формул.

Программисту следует помнить о возможности выхода результата за пределы представления чисел при заданном формате и способе кодирования. В ходе разработки программы такие возможности должны быть исключены, хотя от такого рода ошибок никто не застрахован. Здесь также возможны различные подходы. Например, можно на этапе отладки программы производить контроль результатов с выдачей диагностических сигналов на внешние порты микроконтроллера. Когда программа будет отлажена, то эти блоки контроля можно убрать. Но при использовании такого подхода трудно обеспечить всеобъемлющую проверку программы.

Для контроля выхода результата за пределы разрядной сетки можно использовать биты C и OV регистра PSW. Индикатором выхода за пределы разрядной сетки при сложении и вычитании положительных чисел является установка бита переноса в 1. Бит переполнения устанавливается в 1 в случае переполнения результата при сложении и вычитании чисел со знаком. Его можно также использовать для индикации деления на 0 и выхода произведения за пределы 255. Следует учитывать, что команды умножения и деления записывают в бит переноса 0, поэтому выход результата сложения или вычитания положительных чисел за пределы разрядной сетки нужно проверять сразу после выполнения команд сложения и вычитания. Бит переноса используется для арифметических операций с числами, которые представлены несколькими байтами. Он обеспечивает информационную связь между байтами, поэтому анализ содержимого битов переноса и переполнения нужно производить после вычисления старшего байта результата.

Другой подход основан на ограничении результатов вычислений в случае их переполнения. По физическому смыслу как результаты измерений, так и вычисленные по ним значения управляющих воздействий не могут выходить за определенные пределы. Кроме того, в автоматическом регулировании часто целесообразно использовать ограничение некоторых

управляющих параметров. При использовании второго подхода отладка программы с точки зрения переполнения разделяется на отладку независимых блоков ограничения нескольких параметров. В этом случае после вычисления параметра, значение которого нужно ограничить, следует проверить его текущее значение и в случае необходимости записать в соответствующую ячейку максимальное или минимальное значение.

Рассмотрим простейший случай ограничения положительного значения после выполнения операции сложения:

```
top:  .=     100             ;числовая подстановка для
                             ;максимума
      JC     pma             ;переход по выходу за пределы
                             ;байта
      CJNE   A, #top, chg    ;сравнение с максимальным
                             ;значением
      SJMP   nch             ;переход при максимальном
                             ;значении
chg:  JC     nch             ;переход, если ограничение
                             ;не нужно
pma:  MOV    A, #top         ;запись максимального значения
nch:  NOP                    ;для записи метки
```

В результате выполнения приведенной последовательности команд результат сложения всегда будет находиться в заданных пределах. Если значение top не превышает 255, то приведенный блок может быть использован для ограничения не только после команды сложения, но и после команды прямого счета. Для ограничения результатов сложения и вычитания при использовании отрицательных чисел с записью в дополнительном коде программа немного усложняется. В этом случае значение top должно быть положительным, а значение bot — отрицательным.

```
top:  .=     100             ;числовая подстановка для
                             ;максимума
bot:  .=     -100            ;числовая подстановка для
                             ;минимума
      JNB    OV, chk         ;переход по отсутствию
                             ;переполнения
      JB     A.7, pma        ;переход по больше максимума
      SJMP   pmi             ;переход по меньше минимума
chk:  JB     A.7, neg        ;переход по отрицательному знаку
      CJNE   A, #top, chp    ;проверка по максимуму
      SJMP   nch             ;переход по равенству максимуму
chp:  JC     nch             ;переход, если ограничение не
                             ;нужно
pma;  MOV    A, #top         ;запись максимального значения
```

```
        SJMP    nch             ;на выход из блока
neg:    CJNE    A, #bot, chm    ;проверка по минимуму
chm:    JNC     nch             ;переход, если ограничение не
                                ;нужно
pmi:    MOV     A, #bot         ;запись минимума
nch:    NOP                     ;для записи метки
```

Если значения максимума и минимума расположены не на краю диапазона представления чисел, то этот блок обеспечивает ограничение и после выполнения команд прямого и обратного счета.

Глава 5

Программирование вычисления функций

5.1. Возведение в квадрат и извлечение квадратного корня

За изучением арифметики в школе приступают к началам алгебры, и в дополнение к четырем действиям арифметики ученики знакомятся с возведением в квадрат и извлечением квадратного корня. Поэтому естественно рассмотреть эти алгебраические операции в качестве примера вычисления простейших функций. Начнем с возведения в квадрат числа, занимающего два байта. Запишем формулу квадрата суммы:

```
65536*(X(1)*X(1)) + 256*(2*X(1)*X(0)) + X(0)*X(0).
```

В отличие от перемножения двухбайтовых чисел здесь произведение старшего и младшего байтов нужно вычислять только один раз, поэтому возведение в квадрат выполняется быстрее. Пусть исходное число записано в регистрах R1, R0. Приведем программу, записывающую квадрат этого числа в регистры R3, R2, R1, R0:

```
MOV     A, R1
MOV     B, A
MUL     AB                  ;квадрат ст. байта
MOV     R2, A               ;во 2-й байт квадрата
MOV     R3, B               ;в 3-й байт квадрата
MOV     A, R0
MOV     B, A
MUL     AB                  ;квадрат мл. байта
XCH     R0, A               ;в 0-й байт квадрата
```

```
        XCH     A, B
        XCH     R1, A                ;в 1-й байт квадрата
        MUL     AB                   ;произведение ст. байта на
                                     ;мл. байт
        RLC     A
        XCH     A, B
        RLC     A                    ;произведение удвоено
        JNC     sqr1                 ;переход по отсутствию
                                     ;переноса
        INC     R3                   ;коррекция 3-го байта квадрата
sqr1:   XCH     A, B
        ADD     A, R1
        MOV     R1, A                ;в 1-й байт квадрата
        MOV     A, R2
        ADDC    A, B
        JNC     sqr2                 ;переход по отсутствию
                                     ;переноса
        INC     R3                   ;коррекция 3-го байта квадрата
sqr2:   MOV     R2, A                ;во 2-й байт квадрата
```

За счет записи младших байтов результата в те же регистры, где находилось исходное число, экономятся ресурсы ОЗУ и ПЗУ.

Вычисление обратной функции, то есть извлечение квадратного корня, немного труднее. Обратные функции, как правило, вычисляются труднее, а разрешение некоторых из этих трудностей ведет к новым знаниям. Помня о мнимых числах, будем рассматривать извлечение корня только из положительного числа. Существует несколько алгоритмов непосредственного вычисления квадратного корня. Начнем с метода, основанного на представлении квадрата целого числа N суммой нечетных чисел от 1 до 2*N – 1. Нетрудно убедиться, что разность квадратов соседних чисел всегда нечетна:

```
N*N - (N - 1)*(N - 1) = 2*N - 1.
```

Последовательно вычитая из числа, корень которого требуется определить, нечетные числа 1, 3, 5 и так далее, нужно прекратить вычитание тогда, когда разность станет меньше нуля. Теперь по последнему вычитаемому можно вычислить целую часть значения корня. Пусть в A находится число от 0 до 255. Приведем программу, записывающую целую часть корня от этого числа в накопитель:

```
        CLR     C                    ;подготовка к вычитанию
        MOV     B, #FFh              ;заготовка для вычитаемого
sqrt:   INC     B                    ;увеличение вычитаемого
                                     ;(четное)
        INC     B                    ;увеличение вычитаемого
```

```
                                  ;нечетное)
        SUBB   A, B               ;вычисление разности
        JNC    sqrt               ;повторение по неотрицательной
                                  ;разности
        MOV    A, B               ;вычитаемое
        DEC    A                  ;четное число
        RR     A                  ;корень
```

Недостатком этой программы является существенная зависимость времени вычисления от значения аргумента. Кроме того, вычисленное значение корня всегда меньше истинного, то есть содержит систематическую погрешность.

Рассмотрим пример со значительно лучшими временными характеристиками. В отличие от рассмотренной программы займемся вычислением значения корня с округлением до ближайшего целого. Хорошо известное мнемоническое правило ускоренного вычисления квадрата целого десятичного числа, оканчивающегося на 5, можно переписать в измененном виде:

(N + 0,5)*(N + 0,5) = N*(N + 1) + 0,25.

Это выражение позволяет легко получить ряд граничных значений аргумента, по которым можно определить округленное до целого значение корня простыми сравнениями:

0*1=0, 1*2=2, 2*3=6, 3*4=12, 4*5=20, ...

Если аргумент находится в интервале от 0 до 2 (исключая левую границу и включая правую), то округленное значение корня равно 1, если от 2 до 6, то 2, если от 6 до 12, то 3 и так далее. Притом поиск подходящего интервала можно произвести наискорейшим образом, сравнив аргумент сначала со значением 72. Если он больше, то в 3-ий разряд корня нужно записать единицу, если не больше, то 0. Затем надо сравнить аргумент со значением 156 в первом случае или 20 во втором, что позволяет определить значение второго разряда корня. После третьего и четвертого сравнений определяются соответственно значения первого и нулевого разрядов. Этот метод нахождения подходящего интервала называется поиском по двоичному дереву и широко используется в программировании. Получаемый после завершения поиска номер интервала находится в пределах от 0 до 15, поэтому по завершении поиска для получения числового значения корня к полученному номеру добавляется 1. По сравнению с предыдущим этот алгоритм более чем на порядок уменьшает среднюю погрешность вычислений, хотя максимальная погрешность уменьшается только вдвое.

В приводимой программе аргумент функции записывается в накопитель, а значение корня получается в регистре В:

```
        MOV     B, #0             ;заготовка для корня
        JZ      lt1               ;переход по нулевому корню
        CJNE    A, #8*9, lt8      ;порог 72
        JC      lt8               ;переход по корню меньше 8,5
        ORL     B, #8             ;запись 1 в 3-ий разряд
        CJNE    A, #12*13, lt12   ;порог 156
        JC      lt12              ;переход по корню меньше 12,5
        ORL     B, #4             ;запись 1 во 2-ой разряд
        CJNE    A, #14*15, lt14   ;порог 210
        JC      lt14              ;переход по корню меньше 14,5
        ORL     B, #2             ;запись 1 в 1-ый разряд
        CJNE    A, #15*16, lt15   ;порог 240
        JC      lt15              ;переход по корню меньше 15,5
        SJMP    ge15              ;переход по корню больше или
                                  ;равно 15,5
lt8:    CJNE    A, #4*5, lt4      ;порог 20
        JC      lt4               ;переход по корню меньше 4,5
        ORL     B, #4             ;запись 1 во 2-ой разряд
        CJNE    A, #6*7, lt6      ;порог 42
        JC      lt6               ;переход по корню меньше 6,5
        ORL     B, #2             ;запись 1 в 1-ый разряд
        CJNE    A, #7*8, lt15     ;порог 56
        JC      lt15              ;переход по корню меньше 7,5
        SJMP    ge15              ;переход по корню больше или
                                  ;равно 7,5
lt12:   CJNE    A, #10*11, lt10   ;порог 110
        JC      lt10              ;переход по корню меньше 10,5
        ORL     B, #2             ;запись 1 в 1-ый разряд
        CJNE    A, #11*12, lt15   ;порог 132
        JC      lt15              ;переход по корню меньше 11,5
        SJMP    ge15              ;переход по корню больше или
        равно 11,5
lt14:   CJNE    A, #13*14, lt15   ;порог 182
        JC      lt15              ;переход по корню меньше 13,5
        SJMP    ge15              ;на запись 1 в 0-ой разряд
lt4:    CJNE    A, #2*3, lt2      ;порог 6
        JC      lt2               ;переход по корню меньше 2,5
        ORL     B, #2             ;запись 1 в 1-ый разряд
        CJNE    A, #3*4, lt15     ;порог 12
        JC      lt15              ;переход по корню меньше 3,5
        SJMP    ge15              ;переход по корню больше или
                                  ;равно 3,5
lt6:    CJNE    A, #5*6, lt15     ;порог 30
```

```
        JC      lt15            ;переход по корню меньше 5,5
        SJMP    ge15            ;переход по корню больше или
                                ;равно 5,5
lt10:   CJNE    A, #9*10, lt15  ;порог 90
        JC      lt15            ;переход по корню меньше 9,5
        SJMP    ge15            ;переход по корню больше или
                                ;равно 9,5
lt2:    CJNE    A, #1*2, lt15   ;порог 2
        JC      lt15            ;переход по корню меньше 1,5
ge15:   ORL     B, #1           ;запись 1 в 0-ой разряд
lt15:   INC     B               ;коррекция значения корня
lt1:    NOP                     ;для записи метки
```

В процессе работы программы содержимое накопителя не изменяется. Обратите внимание на то, как записаны значения порогов в командах сравнения. Выражения с операцией умножения перекладывают вычисление пороговых значений на транслятор. Конечно, эта программа занимает гораздо больше места в ПЗУ, нежели предыдущая, но зато в большинстве случаев она и работает быстрее. Из 15 сравнений выполняется только 4, то есть в среднем выполняется только четверть всех команд программы. Этот пример еще раз демонстрирует, что зачастую экономия времени вычисления может быть достигнута за счет затрат объема ПЗУ.

Для вычисления корня из числа, состоящего из двух байтов, можно использовать оба метода следующим образом. Для извлечения корня из старшего байта нужно приспособить второй вариант, изменив значения порогов таким образом, чтобы получить значение корня с недостатком. После вычисления старшей тетрады корня следует вычесть из исходного числа квадрат от приближенного значения корня и затем уточнить младшие цифры вычитанием нечетных чисел, первое из которых вычисляется по приближенному значению корня. Можно также использовать алгоритм извлечения корня "столбиком", которому во время оно обучали в школе. Автор намеренно не касается вычисления корня методом последовательных приближений, так как рекуррентная формула содержит операцию деления. Вычисления корня возможно и табличным методом с помощью интерполяции, но мы рассмотрим табличные методы вычисления далее, для других функций.

5.2. Переход от десятичной системы счисления к двоичной и обратно

При программировании алгоритмов управления или обработки информации могут понадобиться вычисления разнообразных функций.

Иногда целесообразно использовать приближенное представление функции многочленом (полиномом). Рассматривать такой пример в самом общем случае неинтересно. К счастью, задача перехода от одной системы счисления к другой решается при помощи представления чисел в виде полинома. То, что значения коэффициентов полинома (равно как и значения аргумента и функции) целочисленные, в одних случаях не существенно, а в других даже облегчает решение задачи.

При использовании микроконтроллеров для управления аппаратурой программирование большинства вычислений выполняется с представлением чисел в двоичной системе. Это обусловлено двумя причинами. Во-первых, вычисления в двоичных кодах выполняются с наименьшими затратами таких ресурсов, как время и объем оперативной памяти. Во-вторых, десятичное кодирование чисел является обязательным только для обеспечения взаимодействия оператора с аппаратурой и вследствие ограниченного быстродействия человека не требует существенных затрат ресурсов микроконтроллера. Поэтому десятичное кодирование используется только для взаимодействия с оператором, притом вводимые данные подвергаются преобразованию из десятичной системы в двоичную, а выводимые — из двоичной в десятичную. Здесь под двоичной системой может подразумеваться и шестнадцатеричная, так как содержимое одного байта можно представить двумя шестнадцатеричными разрядами без изменения кодирования числа. Двоичное и шестнадцатеричное представления числа "свободно конвертируются" друг в друга, чего не скажешь о двоичном и десятичном.

Для пояснения алгоритмов задачи перехода от одной системы счисления к другой рассмотрим математические выражения, представляющие запись одного и того же числа при основании системы счисления 10 и 16. В первом выражении число записано как M-разрядное с десятичными цифрами D(I), притом разряды нумеруются справа налево, начиная с нулевого. Во втором выражении число записано как N-разрядное с шестнадцатеричными цифрами H(K) и такой же нумерацией разрядов:

```
(((...(H(N-1)*16 + H(N-2))*16 + ...)*16 + H(2))*16 + H(1))*16 + H(0)
(((...(D(M-1)*10 + D(M-2))*10 + ...)*10 + D(2))*10 + D(1))*10 + D(0)
```

В этих формулах основания записаны в десятичиой системе, но формулы верны для любой системы счисления. Количество разрядов в десятичном представлении числа не меньше, чем в шестнадцатеричном (то есть M всегда больше или равно N). Тем, кто знаком с алгеброй, нетрудно понять, что приведенные выражения представляют собой численные значения полиномов степени M–1 и N–1 соответственно. А знающим вычислительную математику известно, что эти выражения записаны с помощью

"схемы Горнера", сводящей к минимуму количество умножений в процедуре вычислений. Постановка задачи перехода от десятичного кодирования числа к шестнадцатеричному такова: для M заданных десятичных цифр D(0), D(1), D(2), ... D(M–2), D(M–1) найти такие N, H(0), H(1), H(2),... H(N–2), H(N–1), чтобы оба выражения имели равные значения.

Эта задача решается весьма просто вычислением второго выражения в системе команд, работающей с двоичными кодами. В следующем примере приведена программа перехода от 4-хразрядного десятичного числа к двоичному. В регистрах R3, R2, R1, R0 записаны двоично-кодированные цифры десятичного числа. Двоичное число получается в регистре B (старший байт) и в накопителе (младший байт).

```
        MOV     A, R3           ;десятичная цифра разряда тысяч
        MOV     B, #10
        MUL     AB
        ADD     A, R2           ;десятичная цифра разряда сотен
        MOV     B, #10
        MUL     AB
        ADD     A, R1           ;десятичная цифра разряда
                                ;десятков
        JNC     dth             ;переход по отсутствию переноса
        INC     B               ;коррекция ст. байта
dth:    MOV     R4, A           ;запоминание мл. байта
        MOV     A, #10
        MUL     AB              ;умножение ст. байта
        XCH     A, R4           ;для умножения мл. байта
        MOV     B, #10          ;
        MUL     AB              ;умножение мл. байта
        ADD     A, R0           ;десятичная цифра разряда единиц
        XCH     A,B
        ADDC    A, R4           ;учет ст. байта произведения
        XCH     A, B
```

По приведенной выше формуле для вычисления двоичного кода нужно умножить три раза, но здесь пришлось использовать 4 команды умножения, так как результат второго умножения может занимать 2 байт, каждый из которых надо умножить на 10.

Обратная задача решалась бы таоже просто — вычислением первого выражения в системе команд, работающей с десятичными кодами. Поскольку в семействе микропроцессоров i8051 таковые практически отсутствуют (об ADD в сочетании с DA серьезно говорить не приходится), необходимо использовать другой способ решения задачи. Здесь уже, по сути дела, приходится решать обратную задачу, когда по значению полинома нужно найти его коэффициенты. Постановка задачи перехода

от шестнадцатеричного кодирования числа к десятичному такова: для N заданных шестнадцатеричных цифр H(0), H(1), H(2),... H(N–2), H(N–1) найти такие M, D(0), D(1), D(2), ... D(M–2), D(M–1), чтобы оба выражения имели равные значения. Для этого вместо умножений на 16 в десятичной системе приходится делить на 10 в шестнадцатеричной. Как видно из второго выражения, остаток от деления на 10 равен десятичной цифре, а частное можно использовать для повторного деления.

Ввиду сложности задачи рассмотрим сначала преобразование числа, занимающего один байт и записанного в накопитель. Программа помещает двоично-кодированные десятичные цифры в регистры R2, R1, R0:

```
MOV     B, #10          ;делитель 10
DIV     AB              ;первое деление
MOV     R0, B           ;цифра единиц
MOV     B, #10          ;делитель 10
DIV     AB              ;второе деление
MOV     R1, B           ;цифра десятков
MOV     R2, A           ;цифра сотен
```

Для получения трех десятичных цифр оказалось достаточным выполнить две команды деления.

С двоичным числом, состоящим из двух байтов, сделать преобразование намного сложнее. Во-первых, число из двух байтов может достигать значения 65535, то есть требуется вычислить 5 десятичных цифр. Во-вторых, при вычислении 3 младших цифр не удается ограничиться одной командой деления на каждую цифру, поскольку делимое может занимать два байта. Приведенную в предыдущей главе программу деления двухбайтового числа нецелесообразно использовать для деления на 10. Вместо этого лучше представить младший байт двумя шестнадцатеричными цифрами, что позволяет использовать остаток от предыдущего деления для вычисления очередного делимого.

Перейдем к программе перевода двоичного числа, записанного в регистры R1, R0, в десятичное. Десятичные цифры должны быть записаны в регистры R4, R3, R2, R1, R0. Для упрощения программы деления выделим отдельные шестнадцатеричные цифры младшего байта. Эти половинки байта не имеют установившегося названия: в русскоязычной литературе для них используется термин тетрада, а в англоязычной — ниббл (nibble).

```
MOV     A, R0           ;мл. байт
MOV     B, #10h         ;константа для выделения
                        ;половинок
DIV     AB              ;выделение половинок
MOV     R0, B           ;мл. половинка мл. байта
XCH     A, R1           ;ст. половинка мл. байта
```

После этой подготовки можно приступить к первому делению, поскольку старший байт уже записан в накопитель. Для вычисления первого частного и первого остатка (цифры единиц) придется использовать три команды деления. После первой команды деления частное не превысит 25, а остаток будет не более 9. Значит частное может занимать более 4 двоичных разрядов. Из остатка и старшей цифры младшего байта нужно составить байт для следующей команды деления. Его значение не может превысить 9*16 + 15 = 159. Частное после второй команды деления укладывается в диапазон представления шестнадцатеричных цифр. Аналогичным образом выполняется подготовка к третьей команде деления. В результате первого деления остаток дает цифру разряда единиц, а частное вновь представлено одним байтом и двумя шестнадцатеричными цифрами.

```
MOV     B, #10
DIV     AB
MOV     R3, A               ;ст. байт 1-го частного
MOV     A, B
SWAP    A
ORL     A, R1
MOV     B, #10
DIV     AB
MOV     R2, A               ;ст. тетрада мл. байта 1-го
                            ;частного
MOV     A, B
SWAP    A
ORL     A, R0
MOV     B, #10
DIV     AB                  ;в А мл. тетрада мл. байта
                            ;1-го частного
MOV     R0, B               ;цифра разряда единиц
```

Следует отметить, что вместо умножения на 16 в этой программе используется команда перестановки половинок байта. Принимая во внимание, что старший байт частного от первого деления записан в регистре R3,   а половинки младшего находятся в R2 и А, выполним второе деление аналогичным образом:

```
XCH     A, R3           ;запись мл. тетрады мл. байта 1-го
                        ;частного
MOV     B, #10
DIV     AB
SWAP    A
MOV     R4, A           ;(мл. тетрада ст. байта 2-го
                        ;частного)*16
MOV     A, B
SWAP    A
```

```
        ORL     A, R2
        MOV     B, #10
        DIV     AB
        ORL     A, R4
        MOV     R4, A           ;объединение двух ст. тетрад 2-го
                                ;частного
        MOV     A, B
        SWAP    A
        ORL     A, R3
        MOV     B, #10
        DIV     AB              ;в А мл. тетрада 2-го частного
        MOV     R1, B           ;цифра разряда десятков
```

На этот раз частное представлено одним байтом в регистре R4 и половинкой байта в накопителе. Поэтому для вычисления третьего частного достаточно двух команд деления:

```
        XCH     A, R4               ;запись мл. тетрады 2-го
                                    ;частного
        MOV     B, #10
        DIV     AB
        SWAP    A
        MOV     R3, A               ;(ст. тетрада 3-го
                                    ;частного)*16
        MOV     A, B
        SWAP    A
        ORL     A, R4
        MOV     B, #10
        DIV     AB                  ;в А мл. тетрада 3-го частного
        MOV     R2, B               ;цифра разряда сотен
```

Старшая половина байта третьего частного находится в R3, а младшая — в накопителе, притом третье частное не превышает 65. Поэтому для вычисления двух старших цифр достаточно одной команды деления:

```
        ORL     A, R3               ;формирование байта 3-го
                                    ;частного
        MOV     B, #10
        DIV     AB
        MOV     R3, B               ;цифра разряда тысяч
        MOV     R4, A               ;цифра разряда десятков тысяч
```

Таким образом, для четырехкратного деления на 10 потребовалось использовать 9 команд деления. Отсутствие операции деления двухбайтового числа на однобайтовое в системе команд микроконтроллера существенно усложняет программу перехода от двоичной системы счисления к десятичной. Ускорение преобразования возможно при использовании таблиц, но этот вариант занимает много места в ПЗУ.

5.3. Вычисление функций при помощи таблиц

Непосредственное вычисление таких функций, как синус, косинус, экспонента и логарифм, или выражений с этими функциями при помощи конечного количества арифметических действий невозможно. В случае необходимости можно вычислять их приближенно, используя аппроксимирующие полиномы со специально подобранными коэффициентами. Разумеется, сложность вычислений зависит от выбранного способа аппроксимации функции, диапазона значений аргументов и требуемой точности. В компьютерах (в частности в математических сопроцессорах) общепринято использование разложений по полиномам Чебышёва, что позволяет хранить для представления этих функций минимальное количество коэффициентов. Но при этом для вычислений используется представление чисел с плавающей запятой.

Для микроконтроллеров альтернативой этому методу является табличное представление функций. Количество значений аргументов, для которых заранее вычисляются значения функции, зависит от ресурсов ПЗУ, поэтому желательно ограничивать размеры таблиц. Вычисление функций для промежуточных значений аргументов производится методом интерполяции. Выбирая размер таблицы и метод интерполяции, необходимо сравнить расход ресурсов микроконтроллера с вариантом использования аппроксимирующих полиномов. Затраты объема ПЗУ на хранение таблицы, как правило, окупаются сокращением объема программы вычисления функции и увеличением ее быстродействия.

Рассмотрим задачу вычисления функции, значения которой заданы в таблице для некоторого количества значений аргументов. Обозначив аргумент и функцию буквами X и Y, запишем таблицу в виде пары столбцов:

```
    X(0)     Y(0)
    X(1)     Y(1)
. . . . . . .
    X(K)     Y(K)
    X(K+1)   Y(K+1)
    . . . . . . .
    X(N-1)   Y(N-1)
    X(N)     Y(N)
```

где индекс определяет номер строки. Обычно данные в таблице располагаются в порядке возрастания значений аргумента, то есть X(K+1) > X(K). В таком виде могут быть записаны и функции, которые заданы математическими выражениями, и экспериментально полученные данные.

Каким образом при помощи таблицы вычисляется значение функции Y для заданного значения аргумента X? Прежде всего нужно проверить, попадает ли значение аргумента в пределы заданной таблицы. Если X < X(0) или X > X(N), то определить значение функции по данной таблице невозможно. Затем нужно найти, наибольшее табличное значение аргумента, не превышающее заданного значения аргумента. Если заданное X совпадает с одним из табличных значений аргумента, то значение функции Y получается непосредственно из таблицы. Рассмотрим случай, когда значение аргумента находится в пределах данной таблицы, но не совпадает ни с одним табличным. Тогда всегда найдется такое K, что X(K) < X < X(K+1). Если функция достаточно гладкая (то есть изменяется достаточно плавно), то для ее оценки при промежуточных значениях аргумента можно использовать вместо реальной зависимости аппроксимирующую кривую, приближенно описывающую эту функцию. Для аппроксимации промежуточных значений функции используются линейная интерполяция и интерполяция по полиномам второй или более высокой степени.

Прежде всего необходимо определить число K, которому соответствует значение аргумента X. Обычно для удобства использования таблицы разность аргументов соседних строк выдерживается постоянной (таблица с фиксированным шагом). В этом случае в программе линейной интерполяции достаточно хранить только таблицу функций, а вместо набора аргументов иметь значение X(0), шаг аргумента D и количество интервалов в таблице N. Для определения номера интервала нужно вычесть из аргумента значение X(0). Если результат меньше нуля, то функция не может быть вычислена. Если результат неотрицательный, то полученную разность нужно разделить на шаг таблицы D, притом частное определяет число K, а остаток — разность (X – X(K)). После деления необходимо проверить, не вышел ли аргумент за верхний предел таблицы. Если частное равно N при ненулевом остатке или больше N, то функция не может быть вычислена. Поиск в таблице для случая непостоянного шага будет рассмотрен позже.

Рассмотрим вычисление функции Y при заданном аргументе X для случая линейной интерполяции. Формула линейной интерполяции основана на равенстве отношения (Y – Y(K)) к (Y(K+1) – Y(K)) отношению (X – X(K)) к (X(K+1) – X(K)). Эта пропорция следует из подобия треугольников с вершинами

```
X(K),Y(K)     X(K+1),Y(K+1)     X(K+1),Y(K)  и
X(K),Y(K)     X,Y               X,Y(K)
```

Отсюда получается интерполяционная формула

```
Y = Y(K) +  ((Y(K+1) - Y(K)) * (X - X(K)))/D
```

Смысл этой формулы в том, что к табличному значению функции, соответствующему аргументу X(K), добавляется приращение функции (конечно, с учетом знака изменения значения функции), пропорциональное отношению отрезков (X – X(K)) и D. Следует обратить внимание на то, что косая черта в интерполяционной формуле обозначает деление с округлением. Команда целочисленного деления, реализующая арифметическое действие деления, вносит систематическую погрешность. Поэтому перед использованием команды целочисленного деления следует добавить к делимому половину делителя.

Таков общий подход к программированию задачи линейной интерполяции таблично заданных функций. Разумеется, детали могут различаться в зависимости от разрядности представления данных в таблице, монотонности функции (приращения только одного знака или разных знаков) и величины шага. Например, при возможности выбора шага целесообразно использовать целую степень двойки, чтобы вместо деления использовать сдвиг (если это экономит время и память).

Практическое применение табличного метода вычисления требует правильного выбора масштаба для представления функции и определения шага таблицы для получения требуемой точности. Рассмотрим в качестве примера задачу вычисления синуса в первой четверти круга:

```
Y = sin X
```

с погрешностью не более 0,005% от максимального значения при использовании линейной интерполяции. Именно с такой погрешностью были представлены значения этой функции в таблицах Брадиса, знакомых многим поколениям российских школьников.

Следует обратить внимание на то, что в общем случае погрешности линейной аппроксимации могут возрастать не только вблизи экстремумов функций, но и в окрестностях особых точек. Поэтому программист должен оценить методические погрешности аналитическим или расчетным способом. Покажем, каким должно быть приращение аргумента таблицы, чтобы погрешность линейной интерполяции не превысила заданного значения. Значение синуса угла, соответствующего середине интервала аппроксимации, равно

```
sin((X(K)+X(K+1)/2)
```

а аппроксимирующее его значение равно половине суммы значений синусов на границах интервала:

```
(sin X(K)+sin X(K+1))/2
```

После тригонометрических преобразований можно записать погрешность линейной аппроксимации в виде

```
2*sin((X(K)+X(K+1)/2)*(sin(D/4))*(sin(D/4))
```

Погрешность линейной аппроксимации пропорциональна квадрату шага таблицы, обозначенному буквой D. Из этого выражения видно, что наибольшая погрешность получается при максимальном значении функции и равна 1/8 квадрата шага, выраженного в радианах. Таким образом, для заданной точности линейной аппроксимации шаг аргумента не должен превышать 0,02 радиана (чуть больше одного градуса). Для вычисления таблицы далее принимается минимальное значение аргумента 0°, максимальное — 90° и шаг — 90°/128.

Перейдем теперь к задаче выбора масштаба, исходя из того, что абсолютная величина синуса не превышает 1. Отведем на хранение каждого значения функции по два байта и определим 7-й бит старшего байта как разряд единиц. Тогда для приведения масштаба произведения к масштабу умножаемой на синус переменной достаточно произвести сдвиг произведения на 1 разряд влево. При вычислении табличных значений нужно каждое полученное десятичное значение синуса умножить на 32768 (2 в 15-й степени), добавить 0.5 и перевести целую часть результата в двоичный код. Таблицу вычисленных таким образом синусов нужно записать в исходный текст программы в следующем виде:

```
      .DATA
tsin: .DW    0192h, 0324h, 0486h, 0647h, 07D9h, 096Bh, 0AFBh, 0C8Ch
      .DW    0E1Ch, 0FABh, 113Ah, 12C8h, 1455h, 15E2h, 176Eh, 18F9h
      .DW    1A83h, 1C0Ch, 1D93h, 1F1Ah, 209Fh, 2224h, 23A7h, 2528h
      .DW    26A8h, 2827h, 29A4h, 2B1Fh, 2C99h, 2E11h, 2F87h, 30FCh
      .DW    326Eh, 33DFh, 354Eh, 36BAh, 3825h, 398Dh, 3AF3h, 3C57h
      .DW    3DB9h, 3F17h, 4073h, 41CEh, 4326h, 447Bh, 45CDh, 471Dh
      .DW    486Ah, 49B4h, 4AFBh, 4C40h, 4D81h, 4EC0h, 4FFBh, 5134h
      .DW    5269h, 539Bh, 54CAh, 55F6h, 571Eh, 5843h, 5964h, 5A82h
      .DW    5B9Dh, 5CB4h, 5DC8h, 5ED7h, 5FE4h, 60ECh, 61F1h, 62F2h
      .DW    63EFh, 64E9h, 65DEh, 66D0h, 67BDh, 68A7h, 698Ch, 6A6Eh
      .DW    6B4Bh, 6C24h, 6CF9h, 6DCAh, 6E97h, 6F5Fh, 7023h, 70E3h
      .DW    719Eh, 7255h, 7308h, 73B6h, 7460h, 7505h, 75A6h, 7642h
      .DW    76D9h, 776Ch, 77FBh, 7885h, 790Ah, 798Ah, 7A06h, 7A7Dh
      .DW    7AEFh, 7B5Dh, 7BC6h, 7C2Ah, 7C89h, 7CE4h, 7D3Ah, 7D8Ah
      .DW    7DD6h, 7E1Eh, 7E60h, 7E9Dh, 7ED6h, 7F0Ah, 7F38h, 7F62h
      .DW    7F87h, 7FA7h, 7FC2h, 7FD9h, 7FEAh, 7FF6h, 7FFEh, 8000h
```

Нулевое значение функции в данном случае можно не хранить. Поэтому нулевому индексу в приведенной таблице соответствует значение аргумента 90°/128. Адреса для соседних значений индексов различаются на 2.

На практике для измерения угла обычно применяется отсчет с цифрового датчика, представляющий собой код Грея. Рассмотрим случай с использованием 12-разрядного датчика угла. Пусть восемь старших разрядов кода записаны в регистре R1, а младшие — в четырех старших разрядах

регистра R0. Показание датчика нужно перевести из рефлексного двоичного кода в позиционный. В отличие от одновременной обработки всех разрядов при переходе от позиционного кода к рефлексному, обратный переход осуществляется поразрядно, начиная со старших разрядов. Самый старший разряд рефлексного и позиционного кодов совпадают. Значения остальных разрядов позиционного кода получается из значений рефлексного кода операцией ИСКЛЮЧАЮЩЕЕ ИЛИ с позиционным кодом соседнего разряда слева.

```
        MOV     R2, #16         ;количество сдвигов при
                                ;преобразовании
        MOV     B, R1           ;ст. байт рефлексного кода
        MOV     A, R0           ;мл. байт рефлексного кода
        CLR     C               ;для записи 0 в мл. разряд
                                ;мл. байта
nxtb:   RLC     A               ;сдвиг мл. байта
        XCH     A, B            ;ст. байт в накопителе
        RLC     A               ;сдвиг ст. байта
        JNC     nchg            ;переход по мл. биту
                                ;позиционного кода
        XRL     A, #80h         ;изменение бита рефлексного
                                ;кода
nchg:   XCH     A, B            ;мл. байт в накопителе
        DJNZ    R2, nxtb        ;переход на продолжение цикла
```

Для вычисления синуса в первом квадранте два старших разряда полученного кода можно отбросить. Оставшаяся часть кода соответствует модулю синуса при любом значении аргумента. Каким образом можно вычислить знаки синуса н косинуса при любом значении аргумента, будет показано далее. Из оставшихся 10 разрядов нужно выделить номер табличного значения и сдвиг значения угла относительно табличного значения аргумента:

```
        ANL     B, #3Fh         ;выделение 6 разрядов ст. байта
        CLR     C               ;для записи 0 в мл. разряд мл.
                                ;байта
        CLR     F0              ;признак ненулевого остатка
        RLC     A               ;получен остаток от деления
                                ;на 128
        JNZ     nzrr            ;переход при ненулевом остатке
        SETB    F0              ;признак нулевого остатка
nzrr:   XCH     A, B            ;ст. байт в накопителе
        RLC     A               ;получено частное от деления на 128
        MOV     R0, B           ;запись остатка в регистр
        DEC     A               ;смещение указателя на начало
                                ;таблицы
```

```
        RLC     A               ;удвоение указателя
        MOV     R1, A           ;запись указателя в регистр
```

Значение частного уменьшается на 1, потому что синус нулевого угла не записан в таблицу. Так как значение функции занимает два байта, значение указателя для выборки из таблицы должно быть удвоено. В случае нулевого значения частного в бит признака заносится 1, что будет использовано для записи нуля на левой границе интервала интерполяции. Для интерполяции в общем случае необходимо прочитать два числа. Так как каждое число состоит из двух байтов, в программе производится 4 выборки из таблицы. Разность соседних значений таблицы может занимать два байта, но она всегда положительна. Это существенно упрощает программу интерполяции:

```
        MOV     DPTR, #tsin     ;загрузка адреса таблицы
                                ;синусов
        JNC     nzra            ;переход, если левая граница не 0
        MOV     R3, #0          ;синус на левой границе равен 0
        MOV     R2, #0          ;
        MOV     R1, #0          ;загрузка указателя правой
                                ;границы
        SJMP    nxtp            ;переход на правую границу
                                ;интервала
nzra:   MOVC    A, @A+DPTR      ;
        MOV     R3, A           ;загрузка ст. байта на левой
                                ;границе
        INC     R1              ;
        MOV     A, R1           ;указатель на следующий байт
        MOVC    A, @A+DPTR      ;
        MOV     R2, A           ;загрузка мл. байта на левой
                                ;границе
nxtp:   JB      F0, ex          ;переход при нулевом остатке
```

Если аргумент соответствует входу в таблицу, то искомое значение синуса уже находится в регистрах R3 и R2. В противном случае нужно прочитать следующее табличное значение:

```
        INC     R1              ;
        MOV     A, R1           ;указатель на следующий байт
        MOV     A, R1           ;указатель на следующий байт
        MOVC    A, @A+DPTR      ;
        XCH     A, R1           ;загрузка ст. байта на правой
                                ;границе
        INC     A               ;указатель на следующий байт
        MOVC    A, @A+DPTR      ;чтение мл. байта на правой
                                ;границе
```

Следующее табличное значение находится в регистре R1 (старший байт) и в накопителе (младший байт). Далее следует линейная аппроксимация для промежуточного значения аргумента. Благодаря выбору шага таблицы в данном случае удалось обойтись без операции деления. Тем не менее следует выполнить округление, учитывая младший байт произведения разности табличных значений и остатка. В противном случае последний бит будет вычисляться с систематической погрешностью.

```
        CLR     C           ;
        SUBB    A, R2       ;мл. байт разности
        XCH     A, R1       ;
        SUBB    A, R3       ;ст. байт разности
        MOV     B, R0       ;
        MUL     AB          ;
        ADD     A, R2       ;к мл. байту от ст. байта разности
        MOV     R2, A       ;
        MOV     A, R3       ;
        ADDC    A, B        ;к ст. байту от ст. байта разности
        MOV     R3, A       ;
        MOV     A, R1       ;
        MOV     B, R0       ;
        MUL     AB          ;
        ADD     A, #80h     ;для округления
        MOV     A, R2       ;
        ADDC    A, B        ;к мл. байту от мл. байта разности
        MOV     R2, A       ;мл. байт синуса
        MOV     A, R3       ;
        ADDC    A, #0       ;учет переноса в ст. байт
        MOV     R3, A       ;ст. байт синуса
ex:     NOP                 ;для записи метки
```

При выбранной точности таблицы погрешность вычисления синуса в основном определяется ограниченной разрядностью датчика.

При помощи этой таблицы можно вычислить косинус угла, используя в качестве входа в таблицу значение аргумента, дополняющее этот угол до 90°. Поскольку при решении прикладных задач тригонометрические функции используются в качестве множителей размерных величин, учитывать выход аргумента за пределы первого квадранта проще после умножения, приписывая необходимый знак произведению. При этом знак синуса задается самым старшим разрядом кода Грея, а знак косинуса — следующим разрядом. Приведенный пример показывает, что для сокращения размеров таблиц целесообразно использовать приведение аргумента функции к минимальному интервалу значений.

5.4. Вычисление обратной функции по таблице прямой функции

В начале главы уже рассматривался частный случай вычисления обратной функции на примере квадратного корня. В общем случае обратную функцию можно вычислять при помощи таблиц так, как показано в предыдущем примере. Но хранение дополнительной таблицы при наличии таблицы для прямой функции связано с дополнительным расходом объема ПЗУ. Поэтому представляте интерес способ использования одной и той же таблицы для вычисления как прямой, так и обратной функции. Решение этой задачи возможно только в случае взаимной однозначности функции и аргумента

Пусть таблица прямой функции имеет постоянный шаг по аргументу. Тогда она является таблицей таких значений аргументов X(K), которые обеспечивают изменение значений обратной функции с заданным шагом

```
D = Y(K+1) - Y(K)
```

Пусть в таблице прямой функции найден наименьший номер строки K, значение которого больше заданного значения аргумента. Тогда искомое значение функции может быть вычислено по интерполяционной формуле

```
Y = D*(K-1) + (D*(X - X(K-1)))/(X(K) - X(K-1))
```

Если вынести D за скобки, то в скобках останется сумма целого числа и правильной дроби. Целая часть определяется поиском интервала в таблице, а дробная часть — интерполяцией. Конечно, программа вычисления обратной функции по таблице прямой функции сложнее рассмотренной ранее программы вычисления прямой функции. Усложнение связано как с поиском табличных значений, так и с более громоздкими вычислениями при интерполяции. Однако в некоторых случаях ее использование может оказаться целесообразным.

Рассмотрим этот способ вычисления обратной функции на примере определения главного значения арксинуса по таблице синусов:

```
Y = arcsin X
```

Главное значение арксинуса определяется как такое значение угла из интервала от минус 90° до плюс 90°, синус которого равен аргументу. Поскольку арксинус является нечетной функцией, достаточно рассмотреть способ вычисления для положительного аргумента. При этом будем предполагать, что аргумент не превышает 1, а в случае равенства аргумента 1 интерполяция не требуется. По определению арксинуса таблица синусов может рассматриваться как таблица аргументов с переменным

шагом, притом табличному значению функции соответствует угол, равный произведению номера строки таблицы на D (в нашем случае 1/128 от прямого угла).

Рассмотрим прежде всего задачу поиска интервала, в котором находится значение аргумента. Для вычисления номера интервала необходимо сравнивать табличные значения прямой функции с заданным значением аргумента обратной функции. Поиск номера интервала простым перебором не представляет никакого интереса ввиду больших затрат времени. Наиболее быстрым способом определения номера интервала является поиск по двоичному дереву. Для этого сначала определяется, к какой половине таблицы относится аргумент. Таким образом определяется старший разряд номера интервала. Затем аналогичным способом определяются все последующие разряды.

. Таблица синусов содержит 128 значений, что очень удобно для использования этого метода. Количество сравнений табличных значений с аргументом не превышает семи. При совпадении табличного значения с аргументом поиск может быть закончен раньше. Приведенная ниже программа поиска интервала использует указатели нижней и верхней границ интервала, значения которых записаны в регистры R1 и R0. Первоначально в эти регистры заносятся минимальное и максимальное значения индексов в массиве таблицы синусов, то есть 0 и 128 соответственно. Пусть старший и младший байты аргумента записаны в ячейках xh и xl соответственно. Значение аргумента сравнивается с табличной выборкой из середины текущего интервала. До окончания поиска сумма значений индексов остается четной, то есть она равна указателю на старший байт очередного табличного значение. Если аргумент равен табличному значению, то поиск заканчивается. Если табличное значение меньше аргумента, то указатель на середину интервала записывается в указатель нижней границы, в противном случае — в указатель верхней границы. Если сумма указателей четная, то повторяется выборка из середины интервала.

Поскольку первое табличное значение соответствует индексу 1, в регистр указателя адреса записывается значение адреса таблицы со смещением –2.

```
        MOV     DPTR, #tsin-2     ;загрузка адреса для поиска в
                                  ;таблице
        MOV     A, #128           ;загрузка указателя на
                                  ;середину таблицы
        MOV     R0, A             ;загрузка индекса конца таблицы
        MOV     R1, #0            ;загрузка индекса начала таблицы
```

```
nxt:  MOV   B, A          ;запоминание указателя
      MOVC  A, @A+DPTR    ;выборка ст. байта из таблицы
      CJNE  A, xh, neq    ;сравнение со ст. байтом аргумента
      MOV   A, B
      INC   A             ;указатель на мл. байт
      MOVC  A, @A+DPTR    ;выборка мл. байта из таблицы
      CJNE  A, xl, neq    ;сравнение с мл. байтом аргумента
```

При совпадении аргумента с табличным значением поиск прекращается, а в регистры R1 и R0 записывается значение функции:

```
      MOV   R1, B      ;значение функции при совпадении
      CLR   A
      AJMP  ex
```

В противном случае в зависимости от результата сравнения изменяется значение верхней или нижней границы и поиск продолжается:

```
neq:  MOV   A, B
      RR    A          ;индекс середины интервала
      JC    ltx        ;переход
      MOV   R0, A      ;середина стала верхней границей
      ADD   A, R1      ;сумма индексов
      SJMP  chke
ltx:  MOV   R1, A      ;середина стала нижней границей
      ADD   A, R0      ;сумма индексов
chke: JNB   A.0, nxt   ;переход при возможности чтения
                       ;таблицы
```

Если сумма индексов нечетная, то поиск завершен.

```
      MOV   A, R1
      JNZ   nza           ;переход по ненулевой левой
                          ;границе
```

Если индекс нижней границы равен нулю, то подготовка к линейной интерполяции упрощается:

```
      MOVC  A, @A+DPTR
      MOV   R3, A
      MOV   A, #1
      MOVC  A, @A+DPTR
      MOV   R2, A
      MOV   R5, xh
      MOV   R4, xl
      SJMP  app          ;переход на линейную интерполяцию
```

При ненулевом индексе нижней границы нужно сделать две выборки из таблицы и вычислить длины двух отрезков:

```
nza:  RL    A
```

```
MOV     B, A
MOVC    A, @A+DPTR          ;выборка ст. байта нижней
                            ;границы
MOV     R3, A
INC     B
MOV     A, B
MOVC    A, @A+DPTR          ;выборка мл. байта нижней
                            ;границы
MOV     R2, A
CLR      C
MOV      A, xl
SUBB     A, R2
MOV      R4, A
MOV      A, xh
SUBB     A, R3
MOV      R5, A              ;числитель дробной части в
                            ;R5, R4
INC     B
MOV     A, B
MOVC    A, @A+DPTR          ;выборка ст. байта верхней
                            ;границы
MOV     R5, A
INC     B
MOV     A, B
MOVC    A, @A+DPTR          ;выборка мл. байта верхней
                            ;границы
CLR      C
SUBB     A, R2
MOV      R2, A
MOV      A, B
SUBB     A, R3
MOV      R3, A              ;знаменатель дробной части в
                            ;R3, R2
```

Остается вычислить значение дробной части. При вычислении дробной части предполагается, что запятая фиксирована перед старшим разрядом байта, записываемого в регистр R0. Здесь следует напомнить, что при вычислении обратной функции при значениях аргумента, близких к экстремуму прямой функции, погрешность результата вычислений существенно возрастает. Из-за того, что в зависимости от значения аргумента значение знаменателя дробной части может разниться более чем в 200 раз, приходится использовать маленькие хитрости. Если знаменатель не превышает 16, то для использования операции целочисленного деления числитель увеличивается в 16 раз:

```
app:    JNZ     hnz         ;по ненулевому ст. байту знаменателя
        MOV     A, R2
        ANL     A, #F0h
        JNZ     gth         ;по нулю в ст. тетраде мл. байта
        MOV     B, R2
        MOV     A, R4
        SWAP    A           ;числитель увеличен в 16 раз
        SJMP    clc
```

В противном случае и числитель, и знаменатель сначала уменьшаются или увеличиваются таким образом, чтобы они были в байтовом формате, в старшем разряде знаменателя была единица:

```
hnz:    RRC     A
        MOV     A, R2
        RRC     A
        MOV     B, A
        MOV     A, R5
        RRC     A
        MOV     A, R4
        RRC     A             ;приведение к байтовому формату
        SJMP    swp
gth:    MOV     A, R2
        MOV     B, R4
chd:    JB      A.7, swp
        RL      A
        XCH     A, B
        RL      A
        XCH     A, B
        SJMP    chd           ;на проверку ст. разряда знаменателя
```

Затем знаменатель уменьшается в 16 раз и производится деление:

```
swp:    ANL     A, #F0h
        SWAP    A
        XCH     A, B
clc:    DIV     AB
        INC     A
        ANL     A, #0Eh       ;округление частного
        SWAP    A             ;перенос в старшую тетраду
```

После этого необходимо привести полученные в регистрах R1 и R0 коды к тому же формату, в котором был представлен аргумент синуса в предыдущей программе:

```
ex:     XCH     A, R1
        RRC     A
        XCH     A, R1
        RRC     A
        MOV     R0, A
```

Аналогичным образом можно запрограммировать и операции деления или извлечения корня как обратные умножению или возведению в квадрат. Например, аналогичную программу можно составить для деления двухбайтового числа на однобайтовое. При этом вместо выборки из таблицы нужно умножать пробное частное (указатель на середину интервала) на делитель и сравнивать полученное произведение с делимым.

5.5. Компенсация систематических погрешностей при помощи таблиц

Характеристики используемых в изделии измерительных и исполнительных устройств, как правило, описываются достаточно простыми математическими соотношениями. Однако в действительности эти характеристики могут отличаться от принятой идеализации. Для существенного повышения точности внешних устройств приходится усложнять технологию их изготовления и отладки и увеличивать производственные затраты. Поэтому целесообразнее решать эту задачу путем калибровки параметров и компенсации погрешностей при обработке информации в микроконтроллере. Не следует забывать, что калибровка и компенсация имеют смысл только в том случае, когда характеристики устройств достаточно стабильны при всех условиях эксплуатации изделия, не изменяются с течением времени. Калибровке и компенсации подлежат только систематические составляющие погрешностей.

Под калибровкой обычно подразумевается компенсация систематических погрешностей, описываемая линейной зависимостью от аргумента. В этом случае говорят о компенсации двух параметров: смещения нуля и крутизны характеристики. В случае нелинейной зависимости приходится учитывать большее количество параметров и производить более сложные вычисления. Если аналитическое выражение для систематической погрешности не удается найти, то эту функцию можно хранить в виде таблицы. Но даже при существовании аналитического выражения табличный метод учета может оказаться более выгодным, так как он позволяет избавиться от решения задачи оценки параметров и от сложных вычислений. При использовании табличного метода вычисления поправок в ПЗУ можно заносить экспериментально полученные таблицы даже в том случае, когда это приходится делать для каждого изделия индивидуально. При необходимости можно записывать калибровочные параметры и таблицы для компенсации в энергонезависимое ЗУ.

Приведем пример программы, позволяющей компенсировать нелинейность характеристики датчика. В данном случае функция Y является экспериментально измеренной систематической погрешностью датчика, то есть разностью между фактическим значением измеряемой величины и показаниями датчика. Предполагается, что показание датчика хранится в двух ячейках памяти — argh, argl (старший и младший байты соответственно), а минимальное значение показания, соответствующее нулевой измеренной величине, и максимальный номер интервала записаны в виде чисел argmih, argmil и n в операндах соответствующих команд (непосредственная адресация). Шаг таблицы выбран таким образом, что старший байт определяет номер интервала, а младший — смещение относительно табличного значения аргумента. Ввиду малости поправок они хранятся в байтовом формате.

```
        CLR     C
        MOV     A, argl
        SUBB    A, #argmil
        MOV     R0, A              ;положение в интервале
        MOV     A, argh
        SUBB    A, #argmih
        JNC     ap1
ap3:    LJMP    err                ;переход по аргументу меньше
                                   ;минимума
ap1:    MOV     R1, A              ;номер интервала
        CJNE    A, #n, ap6
        JC      ap3                ;переход по аргументу больше
                                   ;максимума
```

После определения номера интервала в таблице и положения в интервале нужно сделать две выборки поправок из таблицы. Поскольку поправка может иметь любой знак, этот байт используется для представления чисел от –127 до +127. Для сложения с двухбайтовым числом в случае положительной поправки в старший байт производится запись нулей, в случае отрицательной — единиц.

```
ap6:    MOV     DPTR, #corr        ;адрес таблицы в регистр
                                   ;указателя
        MOVC    A, @A+DPTR         ;поправка для левой границы
                                   ;интервала
        MOV     R2, A              ;поправка на левой границе
                                   ;интервала
        MOV     R3, #0
        JNB     A.7, ap2
        MOV     R3, #FFH           ;отрицательный знак поправки
                                   ;в ст. байт
```

```
ap2:  MOV     A, R1
      INC     A
      MOVC    A, @A+DPTR         ;поправка на правой границе
                                 ;интервала
      CLR     C
      SUBB    A, R2              ;приращение поправки на
                                 ;интервале
```

В данном случае предполагается достаточно медленное изменение поправки. Модуль разности поправок на границах интервала не должен превышать 127. Так как поправка может быть отрицательной, то для интерполяции вычисляются ее модуль и знак. Полученное произведение представляется в первоначальном коде и складывается с поправкой на левом интервале.

```
      MOV     C, A.7
      MOV     F0, C              ;знак приращения
      JNC     ap4                ;переход по положительной
                                 ;разности
      CPL     A
      INC     A                  ;модуль для интерполяции
ap4:  MOV     B, R0
      MUL     AB                 ;учет положения точки в
                                 ;интервале
      ADD     A, #80h            ;для округления ст. байта
      CLR     A
      XCH     A, B
      ADDC    A, #0              ;используется старший байт
      JNB     F0, ap4            ;переход по положительной
                                 ;поправке
      MOV     B, #FFh            ;изменение знака
      CPL     A
      ADD     A, #1
      JNC     ap5
      INC     B
ap5:  ADD     A, R2
      MOV     R2, A
      MOV     A, R3
      ADDC    A, B
      MOV     R3, A              ;вычислена поправка
```

Вычисленная поправка добавляется к показаниям датчика и записывается в регистры R1, R0:

```
      MOV     A, R0
      ADD     A, R2
      MOV     R0, A
```

```
        MOV     A, R1
        ADDC    A, R3
        MOV     R1, A                ;получено точное измерение
```

Для хранения поправок нужно включить в секцию данных массив с директивой инициализации данных:

```
        .DATA
corr:   .DB                          ; поправки
```

В исходный текст можно записывать поправки в десятичном виде со знаком или в двоичном формате.

Если существует более или менее точная математическая модель систематических погрешностей и в процессе настройки изделия удается поучить достоверные оценки параметров, определяющих погрешности, то может оказаться необходимым расчет поправок по формулам этой модели. Когда изделие измеряет векторную величину и существует взаимовлияние погрешностей каналов, то математическая модель будет двумерной. В этом случае расчет по таблицам нецелесообразен, так как их использование для вычисления функций нескольких аргументов требует очень большого объема ПЗУ.

Глава 6

Программирование фильтрации сигналов

6.1. Особенности цифровой фильтрации сигналов

Фильтрацией сигналов называется такая их обработка, в результате которой полностью или частично устраняется влияние шумов и помех на полезный сигнал. С самого начала напомним читателям о двух существенных отличиях цифровой обработки сигнала от аналоговой. Эти отличия связаны с дискретизацией обрабатываемых сигналов по уровню и во времени.

Вследствие дискретизации сигналов во времени, то есть использования конечного количества выборок сигнала на заданном интервале времени, может возникнуть погрешность, обусловленная переносом высокочастотных составляющих сигнала в область низких частот. Ярким примером этой погрешности является стробоскопический эффект, встречающийся в некоторых сценах кинофильмов. Особенно режет глаз такая несообразность, как движение какого-либо экипажа в одну сторону с вращением колес, отвечающим его движению в противоположную сторону. Чтобы при цифровой обработке сигналов не возникали аналогичные эффекты, ширина спектра частот обрабатываемого сигнала не должна превышать половины частоты дискретизации. Поэтому до преобразования входного сигнала в цифровую форму необходимо сначала с помощью аналогового фильтра убрать из сигнала высокочастотные составляющие, не несущие полезной информации.

Преобразование значений аналогового сигнала в цифровые является источником погрешности, зачастую называемой "шумом квантования". Ее влияние особенно существенно при малых амплитудах сигнала. При больших амплитудах эта погрешность может быть оценена на основе следующих допущений относительно ее статистических свойств. Предполагается, что погрешность равномерно распределена на интервале, равном шагу квантования сигнала, и выборки погрешности квантования для всех моментов времени независимы. При этом можно считать, что среднее значение шума квантования равно нулю, а среднеквадратичное отклонение примерно в 3,5 раза меньше шага квантования.

В классической теории фильтров спектр сигнала на выходе фильтра получается умножением спектра входного сигнала на частотную амплитудно-фазовую характеристику фильтра. Преобразования сигнала из временного представления в частотное и обратно даже после изобретения алгоритма быстрого преобразования Фурье (БПФ) требуют больших ресурсов и доступны только специализированным цифровым процессорам сигнала (Digital Signal Processor). Разумеется, в части фильтрации сигналов микроконтроллеры семейства i8051 не могут идти ни в какое сравнение с ними. Но часто возникающие на практике простейшие задачи фильтрации им вполне по силам, если не пользоваться спектральным представлением сигналов. В этом случае вычисление выходного сигнала по значениям входного осуществляется преобразованием, называемым в математике сверткой. Это преобразование известно изучавшим теоретическую электротехнику как интеграл Дюамеля.

Для вычисления интеграла необходимо использовать не только текущее значение сигнала, но и информацию о предыдущих его значениях. Интегрирование обычно производится по формуле прямоугольников. Для линейной фильтрации удается подобрать коэффициенты таким образом, чтобы это упрощение вычислений не вносило методической погрешности. При программирования фильтров обычно используются рекуррентные формулы, позволяющие вычислять выходной сигнал как функцию от текущего входного сигнала и некоторого набора предыдущих входных или выходных сигналов. В общем случае цифровой фильтр должен содержать программно реализованную линию задержки, в которой хранится некоторое количество значений сигналов, предшествующих текущему значению. В зависимости от требуемых характеристик эти значения могут непосредственно пересчитываться в выходной сигнал и/или влиять на значения, записываемые в линию задержки.

В отличие от рассмотренных ранее вычислений, которые должны выполняться как можно быстрее, вычисления при фильтрации должны

выполняться со строго заданным периодом Т, значение которого вкупе с параметрами расчетной формулы влияет на характеристики фильтра. Далее будут приведены примеры программирования линейных фильтров и нелинейного (медианного) фильтра.

6.2. Программирование простейших фильтров нижних частот

Фильтры нижних частот позволяют снизить уровень высокочастотных помех и существенно улучшить отношение сигнал/шум, если мощность шума в полосе частот сигнала достаточно мала. Рассмотрим программирование цифрового фильтра, являющегося аналогом фильтра нижних частот на RC цепочке. Обозначив набор входных сигналов через X(N), а выходных — через Y(N) (здесь N — номер отсчета по времени), запишем рекуррентное соотношение фильтра:

```
Y(N) = (1 - K) * Y(N-1) + K * X(N),
```

где коэффициент $0 < K < 1$ должен быть выбран по желаемой постоянной времени с учетом периода обращения Т к программе фильтрации. Простой проверкой при помощи калькулятора несложно убедиться в том, что при постоянном значении входного сигнала X(N) выходной сигнал Y(N) независимо от его начального значения стремится к величине входного. Если при некотором ненулевом значении выходного сигнала постоянно подавать на вход нулевой входной сигнал, то множество значений выходного сигнала образует убывающую геометрическую прогрессию со знаменателем

```
K = 1 - exp(-T/(R*C)).
```

Для уменьшения количества умножений целесообразно привести рекуррентное соотношение фильтрации к виду

```
Y(N) = Y(N-1) + K * (X(N) - Y(N-1)).
```

Поскольку коэффициент фильтра положительный и не превышает единицы, то в дальнейшем предполагается, что запятая фиксируется перед 7 битом старшего байта коэффициента. При положительных сигналах и допустимости однобайтового представления сигналов и коэффициента программа фильтрации очень проста:

```
        CLR     C               ;подготовка к вычитанию
        MOV     A, x            ;загрузка вх. сигнала
        SUBB    A, y            ;вычитание предыдущего вых.
                                ;сигнала
```

```
        MOV     F0, C           ;запоминание знака разности
        JNC     dp              ;переход по положительной
                                ;разности
        CPL     A               ;вычисление обратного кода
        INC     A               ;получение модуля разности
dp:     MOV     B, k            ;загрузка коэффициента
        MUL     AB              ;вычисление произведения
        JNB     A.7, nc         ;переход, если мл. байт < 1/2
        INC     B               ;добавление 1 в старший байт
nc:     MOV     A, y            ;загрузка предыдущего вых.
                                ;сигнала
        JNB     F0, pos         ;по положительному приращению
        CLR     C               ;подготовка к вычитанию
        SUBB    A, B            ;вычитание модуля приращения
        SJMP    str             ;на запоминание вых. сигнала
pos:    ADD     A, B            ;добавление модуля приращения
str:    MOV     y, A            ;запоминание вых. сигнала
```

Как видно из текста программы, для ее работы используется одна ячейка ОЗУ, в которой между обращениями к программе должно сохраняться значение выходного сигнала фильтра. При необходимости работы с отрицательными значениями сигналов и/или многобайтового представления сигналов и коэффициента необходимо использовать соответствующие приемы программирования, описанные ранее. В соответствии с масштабом коэффициента для вычисления выходного сигнала используется старший байт произведения. Для уменьшения погрешностей вычисления старший байт увеличивается на 1 в том случае, если старший разряд младшего байта произведения равен 1 (округление).

Фильтр нижних частот с одним инерционным элементом называется фильтром первого порядка. Он имеет крутизну частотной характеристики вне полосы пропускания 3 дб на октаву. Приведем пример программирования фильтра нижних частот второго порядка, имеющего два инерционных элемента и крутизну 6 дб на октаву. Разумеется, улучшение характеристик фильтра достигается за счет усложнения алгоритма обработки. Рекуррентное соотношение для выходного сигнала содержит два коэффициента и два предыдущих отсчета сигнала. Для упрощения вычислений целесообразно записать это соотношение так, чтобы кроме предыдущего выходного сигнала запоминать и предыдущее его приращение:

```
Y(N) = Y(N-1) + K1 * (X(N) - Y(N-1)) * + K2 * (Y(N-1) - Y(N-2))
```

Если выбрать граничную частоту полосы пропускания этого фильтра по уровню 3 дб, соответствующую тому же значению произведения RC, что и в предыдущем случае, то выбор коэффициентов задается следующими формулами:

```
K1 = 1 - 2 * exp(-0,7*T/RC) * cos(0,7*T/RC) + exp(-1,4*T/RC)
K2 = exp(-1,4*T/RC)
```

Предполагая, что сигналы положительные однобайтовые и приращение выходного сигнала не превышает 127, можно хранить приращение в виде числа со знаком в ячейке dy.

```
        MOV     A, dy           ;предыдущее приращения вых.
                                ;сигнала
        CLR     F0              ;
        JNB     A.7, dyp        ;по положительному приращению
        SETB    F0              ;запоминание знака приращения
        CPL     A               ;
        INC     A               ;вычисление модуля
dyp:    MOV     B, k2           ;загрузка второго коэффициента
        MUL     AB              ;вычисление произведения
        JNB     A.7, nc2        ;переход, если мл. байт < 1/2
        INC     B               ;добавление 1 в старший байт
nc2:    MOV     A, B            ;запоминание модуля произведения
        JNB     F0, pp          ;по положительному приращению
        CPL     A,              ;
        INC     A               ;вычисление дополнительного кода
pp:     MOV     dy, A           ;временное запоминание произведения
        CLR     C               ;подготовка к вычитанию
        MOV     A, x            ;загрузка вх. сигнала
        SUBB    A, y            ;вычитание предыдущего вых.
                                ;сигнала
        MOV     F0, C           ;запоминание знака разности
        JNC     dp              ;переход по положительной
                                ;разности
        CPL     A               ;
        INC     A               ;вычисление модуля разности
dp:     MOV     B, k1           ;загрузка первого коэффициента
        MUL     AB              ;вычисление произведения
        JNB     A.7, nc1        ;переход, если мл. байт < 1/2
        INC     B               ;добавление 1 в старший байт
nc1:    MOV     A, B            ;для вычисления приращения
        JNB     F0, pos         ;по положительному приращению
        CPL     A               ;
        INC     A               ;вычисление дополнительного кода
pos:    ADD     A, dy           ;вычисление приращения
        MOV     dy, A           ;запоминание приращения
        ADD     A, y            ;добавление предыдущего вых.
                                ;сигнала
        MOV     y, A            ;запоминание вых. сигнала
```

Более сложные программы фильтрации могут быть построены с использованием линий задержки, запоминающих большее количество предыдущих входных или выходных сигналов.

6.3. Программирование фильтра для оценки параметров сигнала

В случае обработки измерений бывает необходимо получить оценку параметров сигнала и вычислить некоторую величину, характеризующую достоверность этой оценки. Рассмотрим простейший фильтр для оценки значения медленно изменяющегося сигнала. Если шум распределен по нормальному закону, то оптимальной оценкой сигнала является его среднее значение на некотором интервале времени. Предполагается, что изменение значения сигнала в течение этого интервала времени достаточно мало по сравнению с допустимой погрешностью оценки. В качестве критерия достоверности оценки воспользуемся суммой квадратов разностей между значениями выборок сигнала и его оценкой.

Чтобы вычислить оценку сигнала и величину, используемую для оценки достоверности, нужно хранить необходимое количество последних выборок сигнала в линии задержки. С этой целью резервируется массив ячеек в ОЗУ, называемый буфером. Для уменьшения размера программы и увеличения ее быстродействия в качестве линии задержки целесообразно использовать следующий порядок записи в массив. В первый раз входной сигнал записывается в начало массива. При каждом последующем обращении к программе фильтрации входной сигнал записывается в следующие ячейки массива до тех пор, пока не будет достигнута верхняя граница массива. После этого запись снова производится в начало массива. Таким образом новый сигнал всегда записывается вместо самого старого, а в буфере всегда находится некоторое заранее заданное количество предыдущих сигналов. При таком способе записи буфер называется кольцевым. Адрес для записи в кольцевой буфер должен сохраняться в отдельной ячейке ОЗУ и изменяться программой фильтрации.

Рассмотрим пример с усреднением по 8 выборкам сигнала, представленным в двухбайтовом формате целыми положительными числами со значениями не более 2000h. Резервирование памяти для кольцевого буфера и для указателя производится директивами

```
      .RSECT
buf:  DS    16
pbuf: DS    1
```

В начале каждого обращения к программе фильтрации нужно записать в фильтр новую выборку на место самой старой. Для этого нужно вычислить новое значение указателя и с помощью косвенной адресации записать в буфер новое значение сигнала. В приведенном примере считается, что в буфере младший байт предшествует старшему, а значение сигнала перед обращением к программе фильтрации было записано в регистры R1, R0.

```
        INC   pbuf          ;приращение указателя
        ANL   pbuf, #07h    ;ограничение приращения
        MOV   A, pbuf       ;для формирования адреса
        RL    A             ;учет двухбайтового формата
        ADD   A, #buf       ;формирование адреса в буфере
        XCH   A, R0         ;загрузка адреса в регистр
        MOV   @R0, A        ;запись мл. байта в буфер
        INC   R0            ;адрес для ст. байта
        MOV   A, R1         ;
        MOV   @R0, A        ;запись ст. байта в буфер
```

После обновления буфера можно суммировать все хранящихся в нем значения сигналов:

```
        CLR   A             ;для накопления мл. байтов
        MOV   B, A          ;для накопления ст. байтов
        MOV   R1, #8        ;для счета количества циклов
        MOV   R0, #buf      ;для чтения из буфера
acum:   ADD   A, @R0        ;накопление мл. байтов
        XCH   A, B          ;
        INC   R0            ;
        ADDC  A, @R0        ;накопление ст. байтов
        XCH   A, B          ;
        INC   R0            ;
        DJNZ  R1, acum      ;на следующий цикл
        MOV   R2, A         ;запоминание мл. байта суммы
        XCH   A, B          ;
        MOV   R3, A         ;запоминание ст. байта суммы
```

Полученная сумма может использоваться в качестве оценки сигнала с наибольшим количеством значащих цифр, а для вычисления суммы квадратов нужно вычислить среднее значение сигнала делением полученной оценки на 8. Это можно сделать посредством трех сдвигов вправо находящейся в накопителе и регистре В суммы, а затем вычислить сумму квадратов отклонений выборок сигнала.

```
        CLR   C             ;
        RRC   A             ;
        XCH   A, B          ;
```

```
        ADD    A, #4          ;округление перед делением
        ANL    A, #F8h        ;очистка лишних разрядов
        RRC    A              ;сумма разделена на 2
        XCH    A, B           ;
        RRC    A              ;
        XCH    A, B           ;
        RRC    A              ;сумма разделена на 4
        XCH    A, B           ;
        RRC    A              ;
        XCH    A, B           ;
        RRC    A              ;сумма разделена на 8
        MOV    R4, A          ;мл. байт среднего значения
        MOV    R5, B          ;ст. байт среднего значения
```

Программу вычисления суммы квадратов следует завершить досрочно по большому отклонению одной из выборок или по большой сумме квадратов.

```
        MOV    R6, #0         ;для накопления суммы квадратов
        MOV    R0, #buf       ;адрес начала буфера
        MOV    R1, #8         ;количество данных в буфере
rms:    CLR    C              ;для вычитания
        MOV    A, @R0         ;чтение мл. байта
        SUBB   A, R4          ;мл. байт разности
        MOV    B, A           ;для возведения в квадрат
        INC    R0             ;адрес ст. байта
        MOV    A, @R0         ;чтение ст. байта
        INC    R0             ;адрес следующего сигнала
        SUBB   A, R5          ;ст. байт разности
        JNC    nmi            ;переход по положительной
                              ;разности
        XRL    B, #FFh        ;для вычисления модуля
        CPL    A              ;для вычисления модуля
        XCH    A, B           ;
        ADD    A, #1          ;модуль мл. байта
        XCH    A, B           ;
        ADDC   A, #0          ;модуль ст. байта
nmi:    JNZ    bad            ;переход по разности больше 255
        MOV    A, B           ;для возведения в квадрат
        MUL    AB             ;квадрат разности
        JB     OV, bad        ;переход по квадрату больше 256
        ADD    A, R6          ;сумма квадратов разности
        JC     bad            ;переход по большой сумме
                              ;квадратов
        DJNZ   R1, rms        ;на следующий цикл
        SJMP   ex             ;на продолжение вычислений
bad:    MOV    R6, #255       ;насыщение суммы квадратов
ex:     NOP                   ;для записи метки
```

В данном случае для накопления суммы квадратов используется только один байт, так как приемлемыми считаются серии измерений с малыми отклонениями. Поэтому при выходе отклонения одной выборки за пределы 15 или при заведомом выходе суммы за пределы 255 программа выходит из цикла и заносит в ячейку с оценкой суммы квадратов максимальное однобайтовое число.

Оценка суммы квадратов отклонений может увеличиваться не только из-за влияния шума. При скачкообразном изменении сигнала его оценка изменяется линейно за время, равное длительности накопления. Во время переходного процесса сумма квадратов отклонений сначала увеличивается, а затем уменьшается до уровня, зависящего от шума. Если сумма квадратов отклонений превышает заранее заданный порог, то для дальнейшей обработки используется старая оценка сигнала. Если сумма квадратов находится в заданных пределах, то используемая для дальнейшей обработки оценка сигнала обновляется.

Интересно сравнить этот алгоритм фильтрации с простым накоплением значения входного сигнала в течение 8 тактов. Приведенной программе для получения первоначальной оценки требуется 8 тактов, а затем оценка выдается каждый такт с запаздыванием 8 тактов. При простом накоплении оценка будет выдаваться только один раз за 8 тактов с тем же запаздыванием. Но при этом нет необходимости запоминания всех значений сигналов. Даже если нужно вычислить сумму квадратов отклонений, то для этого достаточно накопить сумму квадратов значений сигнала. Для накопления суммы квадратов выборок и последующего вычисления суммы квадратов отклонений необходимо работать с промежуточными результатами в форматах до 5 байт.

6.4. Программирование медианного фильтра

В предыдущем примере рассматривалась линейная фильтрация медленно изменяющегося сигнала для достаточно “гладких” помех. В случае редких импульсных помех в сочетании с необходимостью сохранения длительности фронта быстро изменяющегося сигнала бывает целесообразно использовать нелинейную фильтрацию с помощью медианного фильтра. В медианном фильтре обработке подвергается нечетное количество выборок входного сигнала. Из их значений составляется упорядоченный список, а в качестве выходного берется значение из середины списка. Этот способ фильтрации основан на сортировке. Термин медиана

в данном случае относится не к геометрии, а к статистике. Этим словом обозначается параметр статистического распределения, для которого значения одной половины выборок не больше его, а значения другой половины выборок не меньше.

Рассмотрим такой фильтр для случая 5 выборок, имеющих однобайтовый формат. Для сортировки можно записать набор сигналов во вспомогательный массив, а затем произвести их перестановку, чтобы они расположились в убывающем или возрастающем порядке. При этом необязательно полное упорядочение списка, достаточно найти третий по порядку элемент. Другой вариант поиска медианы использует вычисление рангов сигналов, по которым можно определить их положение в списке. При вычислении рангов перестановка не нужна, что экономит время работы программы (оптимизация по быстродействию в этом случае достигается увеличением ее объема). Но в данном случае автор решил привести вариант с вычислением рангов по другим соображениям. До сих пор не представлялось хорошей возможности, чтобы продемонстрировать использование сложных текстовых подстановок. Не испытывая особой склонности к применению таковых, автор все же считает, что они весьма эффективны при программировании некоторых алгоритмов.

Пусть входной сигнал находится в ячейке x, а выходной сигнал должен быть записан в ячейку y. В предыдущем примере текущий сигнал записывался в буфер перед началом фильтрации, что увеличивает затраты объема ОЗУ. На самом деле в буфере нужно хранить только предшествующие выборки входного сигнала. Для этого надо обновлять содержимое буфера только после обработки вновь поступившего сигнала. В данном случае между обращениями к программе нужно сохранять один байт в качестве указателя самого старого сигнала и четыре байта в качестве значений предыдущих сигналов. Зададим имена каждой из ячеек кольцевого буфера следующим образом:

```
       .RSECT
pbuf:  .DS     1              ;указатель на кольцевой буфер
buf0:  .DS     1              ;ранг записывается в R4
buf1:  .DS     1              ;ранг записывается в R5
buf2:  .DS     1              ;ранг записывается в R6
buf3:  .DS     1              ;ранг записывается в R7
```

Значения рангов (информации о старшинстве сигналов) будем накапливать в регистрах общего назначения. Предназначим регистр R3 для ранга входного сигнала, а регистры R4, R5, R6 и R7 для рангов сигналов, хранящихся в буфере. В начале работы программы в эти регистры заносятся нули. Для подсчета рангов пяти сигналов нужно произвести сравнение

10 пар значений (аналогично проведению турнира по круговой системе). При равенстве значений сигналов содержимое соответствующих им регистров не изменяется. При неравенстве в регистр большего сигнала добавляется 1, а из регистра меньшего сигнала вычитается 1. Определим операцию накопления ранга с двумя формальными параметрами, фактические значения которых должны соответствовать номерам используемых регистров:

```
        .CODE
RANK:   .MACRO f,s
        CJNE   A, buf|<s-4>, ch#
        SJMP   ex#
ch#:    JC     lt#
        INC    R|f
        DEC    R|s
        SJMP   ex#
lt#:    DEC    R|f
        INC    R|s
ex#:
        .ENDM
```

Перед использованием этой операции в накопитель нужно занести сигнал, соответствующий первому фактическому параметру. В регистровых операндах команд счета используется подстановка символа фактического параметра, что обозначено операцией конкатенации (вертикальная черта). Во втором операнде команды сравнения в результате подстановки должно быть записано имя ячейки буфера. Перед подстановкой приходится выполнять арифметическую операцию вычитания. Необходимость вычисления выражения до подстановки обозначена угловыми скобками. Все метки в операции подсчета рангов локальные. При равенстве сигналов выполняется 2 команды. Если первый сигнал меньше второго, то выполняется 4 команды. Если первый сигнал больше второго, то выполняется 6 команд.

Запишем часть программы, осуществляющую подсчет рангов. За счет применения сложной текстовой подстановки исходный текст для подсчета рангов получился компактным и легко читаемым:

```
      MOV    A, #0
      MOV    R3, A
      MOV    R4, A
      MOV    R5, A
      MOV    R6, A
      MOV    R7, A              ;запись нулей в счетчики ранга
      MOV    A, x               ;для проверки входного сигнала
      RANK   3,4
```

```
RANK    3,5
RANK    3,6
RANK    3,7
MOV     A, buf0          ;для проверки 0-ой ячейки буфера
RANK    4,5
RANK    4,6
RANK    4,7
MOV     A, buf1          ;для проверки 1-ой ячейки буфера
RANK    5, 6
RANK    6, 7
MOV     A, buf2          ;для проверки 2-ой ячейки буфера
RANK    6, 7
```

Но главное достоинство использования сложной текстовой подстановки заключается в том, что вся формальная работа по записи меток и операндов осуществляется ассемблером. Транслятор преобразует приведенные 20 строк исходного текста в 100 и переведет их в 100 машинных команд с 30 метками.

Теперь по содержимому регистров нужно определить, в какой ячейке находится медиана. Если бы все сигналы были разные, то все ранги также были бы разными. Поэтому медиану можно было бы найти по регистру с нулевым содержимым. В случае совпадения значений сигналов поиск медианы несколько усложняется. Рассмотрим все варианты распределения рангов, расположенных по старшинству:

```
 0, 0, 0, 0, 0     значения всех сигналов совпадают
+1,+1,+1,+1,-4
+2,+2,+2,-3,-3
+2,+2,+2,-2,-4
+3,+3,-2,-2,-2
+3,+3,-1,-1,-4
+3,+3, 0,-3,-3
+3,+3, 0,-2,-4
+4,-1,-1,-1,-1
+4, 0, 0, 0,-4
+4,+1,+1,-3,-3
+4,+1,+1,-2,-4
+4,+2,-2,-2,-2
+4,+2,-1,-1,-4
+4,+2, 0,-3,-3
+4,+2, 0,-2,-4     значения всех сигналов различны
```

Анализ всех 16 вариантов показывает, что ранг с модулем не более 1 всегда соответствует медиане. Если минимальный модуль ранга равен 2, то медиане соответствует ранг с таким знаком, какой имеется еще у двух таких же значений ранга. Ранги с модулями 3 и 4 никогда не соответствуют медиане.

Для определения номера регистра, удовлетворяющего заданным требованиям, используется сложная текстовая подстановка с одним формальным параметром. Если в результате проверки удалось вычислить номер регистра, соответствующего медиане, то управление передается на нелокальную метку в ту часть программы, которая должна записать медиану в ячейку у:

```
MDN:   .MACRO n
       MOV    R0, #|n
       MOV    A, R|n
       JZ     ldm
       JNB    A.7, pls#
       CPL    A
       SJMP   ch1#
pls#:  DEC    A
ch1#:  JZ     ldm
       DEC    A
       JNZ    nxt#
       MOV    A, R|n
       JB     A.7, ch2#
       INC    R1
       CJNE   R1, #3, nxt#
       SJMP   ldm
       SJMP   nxt#
ch2#:  INC    R2
       CJNE   R2, #3, nxt#
       SJMP   ldm
nxt#:
       .ENDM
```

Сначала на случай обнаружения подходящего ранга в регистр R0 заносится номер проверяемого регистра. Затем содержимое проверяемого регистра пересылается в накопитель. В случае нулевого ранга управление передается на загрузку медианы. В противном случае вычисляется модуль ранга, уменьшенный на единицу. Если модуль ранга равен 1, то управление передается на загрузку медианы. Если он больше 2, то управление передается на последнюю метку подстановки. Для рангов +2 и –2 в регистрах R1 и R2 подсчитывается количество соответствующих им сигналов. Если содержимое счетчика оказалось равным 3, управление передается на загрузку медианы. Время выполнения подстановки зависит от содержимого проверяемых регистров. При нулевом ранге выполняется 3 команды, а при рангах +1 и –1 — 6 и 7 команд соответственно. При рангах +3 и +4 выполняется 8 команд, а при рангах –3 и –4 выполняется 9 команд. При модуле ранга, равном 2, выполняется 14 команд.

Определившись с подстановкой для поиска медианы, можно написать завершающую часть программы фильтрации:

```
MOV     R1, #0          ;для подсчета количества рангов +2
MOV     R2, #0          ;для подсчета количества рангов -2
MDN     3
MDN     4
MDN     5
MDN     6
MOV     R0, #7          ;проверка рангов завершена
```

Длительность поиска медианы существенно зависит от расположения выборок в массиве. В наилучшем случае, когда нулевой ранг содержится в первом проверяемом регистре, поиск займет 3 команды. Если нулевой ранг содержится в последнем проверяемом регистре, то для поиска медианы придется затратить свыше 40 команд. Наиболее длительным оказывается поиск медианы при трех совпавших сигналах, когда модуль ранга равен 2. Обратите внимание на то, как по номеру регистра вычисляется адрес передачи управления на команду записи медианы в ячейку y.

```
ldm:  MOV     A, R0          ;номер регистра
      MOV     B, #5          ;длина блока загрузки медианы
      MUL     AB             ;для вычисления адреса перехода
      SUBB    A, #15         ;смещение относительно адреса ldy
      MOV     DPTR, #ldy
      JMP     @A+DPTR
ldy:  MOV     y, x
      SJMP    ldx
      MOV     y, buf0
      SJMP    ldx
      MOV     y, buf1
      SJMP    ldx
      MOV     y, buf2
      SJMP    ldx
      MOV     y, buf3
```

В завершение программы производится запись входного сигнала в буфер:

```
ldx:  INC     pbuf
      ANL     pbuf, #03h
      MOV     A, pbuf
      ADD     A, #buf0
      MOV     R0, A
      MOV     @R0, x        ;запись входного сигнала в буфер
```

Вариант с вычислением рангов имеет явные преимущества перед вариантом с сортировкой в том случае, если значения сигнала представлены в двухбайтовом формате. Увеличение формата удваивает количество

команд, используемых для пересылки сигналов. Подсчет рангов усложняется совсем немного, а блок поиска медианы можно использовать без изменений.

Приведенными примерами не исчерпываются варианты программирования медианного фильтра. Можно запоминать ранги сигналов, чтобы уменьшить количество операций сравнения за счет их коррекции при замене старого сигнала на новый. Можно сохранять массив сигналов в отсортированном виде для упрощения поиска медианы. Но в этом случае необходимо хранить и сортировать одновременно с сигналами их атрибуты, указывающие на время их хранения в соответствующих массивах.

Глава 7

Программирование взаимодействия с внешними устройствами

7.1. Общие вопросы взаимодействия

При программировании алгоритмов решения вычислительных задач необходимо учитывать следующие ресурсы микроконтроллера: систему команд, форматы данных, допустимый расход объемов ОЗУ и ПЗУ. Задача программирования взаимодействия с внешними устройствами требует учета ряда дополнительных ограничений. Часто самым существенным ограничением является время решения задачи управления изделием и/или время реакции на внешние сигналы. Для взаимодействия микроконтроллера с внешними устройствами нужно обеспечить выполнение некоторых наборов требований, которые в вычислительной технике называются интерфейсами. Часть этих требований относится к конструкторской и схемотехнической стороне разработки изделия. Предметом данной главы является обеспечение той части требований, которые относятся к программированию.

Задачей программиста является обеспечение логической части интерфейса, заключающейся в выдаче и/или приеме некоторых последовательностей двоичных кодов в определенном масштабе времени и в зависимости от выполнения тех или иных условий. Двоичные коды могут передаваться параллельно (разные биты по разным проводам) или последовательно (все биты по одному проводу). Последовательная передача может начинаться с младших или со старших битов кода. Для экономии времени может осуществляться дуплексная связь, обеспечивающая

одновременную передачу и прием по разным линиям связи. Обмен кодами предусматривает не только прием и передачу сигналов, но и отношения подчинения между устройствами. Одно из устройств должно быть ведущим, а другое — ведомым. Для обеспечения достоверности программа должна обеспечивать проверку принимаемых сигналов на соответствие контрольным кодам и выдачу контрольных кодов для проверки передаваемых сигналов в месте приема. В случае обнаружения сбоев при приеме программа должна запрашивать повторную выдачу информации, а при передаче повторять выдачу информации по запросу. Для уменьшения количества линий связи может использоваться мультиплексирование, обеспечивающее обмен с несколькими устройствами по одним и тем же линиям связи. В этом случае сеанс связи со стороны ведущего устройства должен начинаться с выдачи идентификационного кода ведомого устройства. В ответ на этот код соответствующее ведомое устройство должно выдать ведущему устройству сигнал готовности к сеансу. Приведенный (далеко не полный) перечень возможных требований к логической части интерфейса показывает, что программисту есть над чем поработать. Существуют стандартные способы обмена сигналами, описания которых называются протоколами, но мы не будем рассматривать ни эти протоколы, ни их программирование.

Для микроконтроллеров семейства i8051 разработано множество разнообразных наборов микросхем (chip set), обеспечивающих различные прикладные задачи. Можно самостоятельно разрабатывать программы для взаимодействия микроконтроллера с этими микросхемами. Но в отличие от программирования чисто вычислительных задач в данном случае требуются дополнительные знания о логике работы этих устройств. Если есть возможность, то лучше использовать отработанные программы для взаимодействия микроконтроллеров с микросхемами, которые можно назвать драйверами в соответствии с применяемой для компьютеров терминологией. В связи с многочисленностью типов микросхем и разнообразием их применений рассмотрение вопроса программирования драйверов заслуживает отдельной книги. По этой причине в настоящей главе затронуты только наиболее существенные направления работы, которую должен проделать программист, чтобы обеспечить взаимодействие микроконтроллера с остальными частями изделия и с оператором. Приведенные примеры практически не касаются подробностей аппаратной реализации внешних устройств. Здесь мы остановимся только на программировании простейших задач ввода/вывода без использования дополнительных микросхем.

В первую очередь необходимо разобраться, каким образом организовать работу программы в реальном масштабе времени. Для этого программа должна обеспечить определенную цикличность выполнения всех задач во времени и оперативность реакции на внешние сигналы. Выполнение каждой из команд микроконтроллером занимает интервал времени, кратный периоду колебаний в задающем генераторе. Поэтому можно рассчитать длительности работы различных блоков программы. Теоретически возможно обеспечить синхронизацию работы программы посредством учета затрат времени на выполнение команд с последующим выполнением некоторого количества пустых команд, но о практических примерах такого рода слышать не приходилось. Для реакции на внешние сигналы можно запрограммировать периодическую проверку его наличия на соответствующем входе микроконтроллера. Но слишком большая частота опроса входа ухудшает производительность программы, а при малой частоте опроса время реакции на внешний сигнал будет большим. Для синхронизации работы программ и обработки внешних запросов в вычислительной технике используются прерывания (interrupt), которые позволяют обеспечить быструю реакцию на события без снижения производительности программы.

7.2. Порядок выполнения прерываний в микроконтроллерах семейства i8051.

Механизм прерываний разработан для общения вычислительных устройств с внешним миром. Прерыванием называется выполнение определенных команд не в порядке выполнения программы, а по некоторому событию, называемому запросом на прерывание. Обработка запроса на прерывание откладывается до завершения текущей команды прерываемой программы. Затем производится проверка возможности выполнения прерывания. Если данный вид прерывания разрешен, то проверяется приоритет запроса на прерывание. Если запрос пришел во время обработки прерывания с более высоким приоритетом, то его исполнение откладывается (но не отменяется). Управление приоритетами прерываний и разрешением прерываний производится программно. Команды прерывания не входят в набор команд микроконтроллера, используемых программистом. Логически они являются аналогами команды дальнего перехода к подпрограмме и передают управление на определенный адрес,

заданный архитектурой микроконтроллера. Выполнение запроса на прерывание начинается с запоминания в стеке адреса той команды, выполнение которой откладывается для обработки прерывания. Затем управление передается по адресу, соответствующему виду прерывания. В эту ячейку программист должен записать первую команду программы обработки прерывания. Использование прерываний позволяет отделить программирование задач синхронизации и обработки запросов от программирования вычислительных задач.

Микроконтроллеры семейства i8051 работают не менее чем с 5 видами прерываний. Два вида прерывания производятся по внешним сигналам (INT0, INT1), два — по переполнению счетчиков (T/C0, T/C1) и один — по завершению приема или передачи байта через последовательный порт. Старшие модели микроконтроллеров имеют больше видов прерываний. Каждому из дополнительных видов прерываний отводится свой начальный адрес в ПЗУ, называемый по аналогии с компьютерами вектором прерывания.

Далее будут рассмотрены только примеры обработки прерывания для синхронизации работы программы при помощи внешнего сигнала и при помощи счетчиков микроконтроллера. Но ради полноты изложения приведем фиксированные адреса ПЗУ, по которым должны располагаться первые команды блоков программы с определенным функциональным назначением:

```
0000h     ;первая команда выполняемая после включения
0003h     ;первая команда обработки прерывания по INT0
000Bh     ;первая команда обработки прерывания по T/C0
0013h     ;первая команда обработки прерывания по INT1
001Bh     ;первая команда обработки прерывания по T/C1
0023h     ;первая команда обработки прерывания по посл. порту
```

Отведенное на эти команды адресное пространство (3 байт для инициализации и по 8 байт для обработки прерываний) как правило используется только для записи команд передачи управления, потому что оно слишком мало для размещения соответствующих блоков программы. Программа обработки прерывания должна заканчиваться командой возврата из прерывания. По этой команде осуществляется возврат к исполнению прерванной программы.

Разрешение на обработку прерываний осуществляется установкой в единицу соответствующих битов регистра разрешения прерывания IE (Interrupt Enable):

```
IE.7 = EA  (Enable All)          все прерывания
```

```
IE.4 = ES  (Enable Serial)      прерывание по последователь-
                                ному каналу
IE.3 = ET1 (Enable Timer 1)     прерывание по таймеру 1
IE.2 = EX1 (Enable eXternal 1) прерывание по внешнему входу 1
IE.1 = ET0 (Enable Timer 0)     прерывание по таймеру 0
IE.0 = EX0 (Enable eXternal 0) прерывание по внешнему входу 0
```

Запись нуля в старший бит этого регистра запрещает обработку всех прерываний, а единица в этом бите разрешает только проверку разрешений в остальных битах регистра. При прочих равных условиях приоритет имеют прерывания с меньшим номером бита. В случае необходимости можно разделить приоритеты всех сигналов прерывания на две группы при помощи записи единиц и нулей в соответствующие биты регистра приоритетов прерывания IP (Interrupt Priority):

```
IP.4 = PS  (Priority Serial)      приоритет последовательно-
                                  го канала
IP.3 = PT1 (Priority Timer 1)     приоритет таймера 1
IP.2 = PX1 (Priority eXternal 1) приоритет внешнего входа 1
IP.1 = PT0 (Priority Timer 0)     приоритет таймера 0
IP.0 = PX0 (Priority eXternal 0) приоритет внешнего входа 0
```

Группа с единицами в битах приоритета проверяется в первую очередь, а группа с нулями проверяется во вторую, причем внутри группы предпочтение отдается прерыванию с меньшим номером бита. В старших моделях для управления разрешениями прерываний и их приоритетом используются биты в тех же или в дополнительных регистрах.

7.3. Синхронизация работы программы внешним или внутренним сигналом

Все ранее рассмотренные примеры программ относятся к решению частных задач обработки информации. Микроконтроллеру приходится выполнять множество программ, решающих частные задачи. Поскольку одновременное выполнение программ невозможно, необходимо обеспечить некоторую последовательность решения задач посредством объединения программ в одну общую программу. При объединении программ, решающих частные задачи, в общую программу необходимо синхронизировать их работу. Для этого нужно разработать синхронизирующую программу, которая должна обеспечить определенные последовательность и периодичность выполнения программ, решающих частные задачи,

в зависимости от их функционального назначения программ и информационных связей между ними. По аналогии с непрерывным и ждущим режимами развертки осциллоскопа существует два вида синхронизации. В первом случае необходимо повторять решение частной задачи с определенным периодом, который задает масштаб времени для соответствующего процесса управления. Во втором случае частная задача должна решаться однократно для каждого из внешних запросов, инициирующих ее выполнение. Программирование задач управления реальными объектами интереснее и труднее программирования чисто вычислительных задач. Тем, кто привык к программированию чисто вычислительных задач, нужно освоить разработку синхронизирующих программ.

Большинство задач, выполняемых программой управления изделием, должно решаться в рамках строгих ограничений по времени, но решение некоторых задач можно откладывать, отдавая приоритет решению срочных задач. Назовем те задачи, решение которых можно откладывать, фоновыми. Синхронизирующая программа как бы "нарезает" отрезки времени и предоставляет их программам, решающим срочные задачи. После решения срочной задачи сумма оставшихся частей каждого из этих отрезков времени предоставляется программам, решающим фоновые задачи. При использовании более чем одного вида прерывания управление разрешениями прерываний и назначениями приоритетов усложняется в связи с возможностью конфликтов и необходимостью обслуживания очередей прерываний. Общего рецепта для обслуживания прерываний не существует. Однако навык занимать места в нескольких очередях одновременно позволяет российским программистам решать такие задачи не только по закону, но и по справедливости.

Для простоты изложения рассмотрим далее программирование обработки прерываний только с точки зрения обеспечения заданного масштаба времени. Период цикла прерываний определяется наивысшей из частот необходимых для обмена с другим устройством или с управляемым изделием. С целью упрощения синхронизирующей программы все периоды для других обменов с ведомыми устройствами следует делать кратными периоду цикла прерываний, хотя это и необязательно. Для распределения ресурсов по времени решения задач нужно составить временную диаграмму синхронизации для цикла с наиболее низкой частотой. Отдельные части этого цикла, распределяемые между частными задачами, можно назвать фазами.

При разработке синхронизирующей программы нужно составить расписание очередности решения срочных задач с распределением их по фазам работы этой программы. Затем нужно оценить, в какой фазе работы

получаются наибольшие затраты времени решения задач. При этом следует учитывать ситуации, когда запросы от других устройств (как ведущих, так и ведомых) могут прийти в одном и том же цикле прерывания. Если время решения превышает длительность фазы, то нужно попытаться перераспределить решение задач между фазами с целью устранения перегрузки. Если это не удается, то нужно разработать программы решения задач с большим быстродействием или использовать более быстродействующий процессор.

Примем за начало периода работы программы выполнение первой команды обработки запроса на прерывание. Каждый период используется для выполнения срочных задач, а на фоновые задачи отводится время от завершения срочных задач до начала следующего периода. Таким образом срочные задачи должны выполняться как обработка периодических запросов на прерывание. Фоновые задачи обычно выполняются по запросам от оператора, которые можно считать случайными. Поэтому требования к времени их выполнения менее строгие. Необходимо обеспечить завершение фоновой задачи до появления следующего запроса. В противном случае необходимо создавать очередь обслуживания запросов, что еще более усложнит синхронизирующую программу.

Программу синхронизации нельзя рассматривать в отрыве от программы инициализации, которая должна быть выполнена при включении изделия. Программа инициализации записывает исходную информацию в функциональные регистры процессора и в рабочие ячейки ОЗУ, задает режимы работы внешних устройств, а также (в случае необходимости) читает информацию из энергонезависимого ЗУ. Только после этого можно перейти к выполнению программы синхронизации, обеспечивающей распределение времени между срочными и фоновыми задачами. Как правильно разместить блок обработки периодических прерываний, включающий в себя программы выполнения срочных задач, и программы обработки фоновых задач? Блок обработки периодических прерываний должен в первую очередь обеспечить своевременное прерывание для следующего цикла. После этого необходимо выполнить срочные задачи очередного цикла, по завершении которых нужно вернуться к решению фоновых задач. Поскольку и фоновая, и инициализирующая части программы выполняются не в состоянии периодического прерывания, после инициализации управление должно быть передано фоновой части программы. Приведем пример размещения перечисленных блоков программы в случае использования внешнего сигнала синхронизации.

Синхронизация прерыванием от внешнего сигнала может осуществляться посредством подключения к входу INT0 импульсного сигнала

нужной частоты с двумя уровнями, соответствующими логическим 0 и 1 сигналов микроконтроллера. В простейшем случае он может быть сформирован из напряжения сети (частота 50 Гц) или от двухполупериодного выпрямителя до фильтрации напряжения (100 Гц). Такие частоты прерываний приемлемы для многих применений микроконтроллеров, хотя из-за помех в сети длительность интервала между прерываниями нестабильна. Если тактовая частота микроконтроллера задана кварцем на 12 МГц, то в первом случае за один цикл синхронизации с длительностью 20 мс может быть выполнено 20000 коротких команд, а во втором за один цикл с длительностью 10 мс — 10000. Для запроса прерывания по переходу входного сигнала из 1 в 0 необходимо записать единицу в бит TCON.0. Блоки программы могут быть расположены следующим образом:

```
       .CODE
       .ORG    00h       ;адрес первой выполняемой команды
       JMP     init      ;переход на блок инициализации
       .ORG    03h       ;адрес команды для прерывания по INT0
       JMP     main      ;переход на основную часть программы
       .ORG    30h       ;адрес блока инициализации
init:                    ;начало блока инициализации
            ;запись кодов в порты микроконтроллера
            ;установка режимов работы внешних устройств
            ;запись кодов в функциональные регистры
            ;чтение исходных данных из энергонезависимого ЗУ
            ;запись начальных значений в рабочие ячейки
       SETB    TCON.0      ;установка режима прерываний по INT0
       MOV     IE, #01h    ;разрешение прерываний по INT0
                  ;конец блока инициализации
bckg:                      ;начало фоновой части программы
                  ;выполнение фоновых задач
       SJMP    bckg        ;бесконечный цикл
main:                      ;начало основной части программы
       CLR     EA          ;запрет прерываний
                  ;сохранение данных фоновой задачи
                  ;выполнение срочных задач
                  ;восстановление данных фоновой задачи
       SETB    EA          ;разрешение прерываний
       RETI                ;возврат к фоновой части программы
```

При синхронизации прерыванием от внешнего сигнала подготовка к следующему циклу не нужна. Обратите внимание, что в самом начале программы обработки прерываний необходимо сохранить содержимое всех тех функциональных регистров, которые используются и основной, и фоновой частями программы. Перед выходом из основной программы нужно восстановить содержимое этих регистров.

Более широкие возможности для управления длительностью цикла прерываний предоставляют внутренние таймеры-счетчики микроконтроллера. Рассмотрим синхронизацию при помощи таймера-счетчика с номером 0. В этом случае в блоках инициализации и подготовки к следующему прерыванию необходимо записывать в таймер определенные коды. Для определенности зададимся тактовой частотой микроконтроллера 12 МГц и частотой прерывания 100 Гц. В отличие от предыдущего примера при инициализации в регистр разрешения прерываний и в регистр управления режимами записываются другие коды; кроме того, необходимо инициализировать регистр управления счетчиками. Буква X в записываемом в этот регистр коде обозначает, что старшая шестнадцатеричная цифра должна быть выбрана в соответствии с необходимым режимом работы счетчика 1. В этом примере коды для записи в счетчик выбраны таким образом, чтобы первое прерывание после инициализации произошло через 256 мкс после завершения инициализации, а период прерываний составил 10 мс:

```
        .CODE
        .ORG   00h              ;адрес первой выполняемой команды
        JMP    init             ;переход на блок инициализации
        .ORG   0Bh              ;адрес команды для прерывания по T0
        CLR    EA               ;запрет всех прерываний
        JMP    main             ;переход на основную часть программы
        .ORG   30h              ;адрес блока инициализации
init:                           ;начало блока инициализации
           ;запись кодов в порты микроконтроллера
           ;установка режимов работы внешних устройств
           ;запись кодов в функциональные регистры
           ;чтение исходных данных из энергонезависимого ЗУ
           ;запись начальных значений в рабочие ячейки
        MOV    TMOD, #X9h       ;включение счетчика 0 в режиме таймера
        MOV    TH0, #FFh        ;запись кодов начальной задержки
        MOV    TL0, #00h        ;в счетчик
        MOV    IE, #82h         ;разрешение прерываний по входу T0
        SETB   TR0              ;запуск счетчика T0
                      ;конец блока инициализации
bckg:                           ;начало фоновой части программы
                      ;выполнение фоновых задач
        SJMP   bckg             ;бесконечный цикл
main:                           ;начало блока прерывания
        CLR    EA               ;запрет прерываний
        MOV    TH0, #D9h        ;ст. байт доп. кода периода цикла
        MOV    TL0, #05h        ;мл. байт доп. кода периода цикла
        ;сохранение данных фоновой задачи
```

```
                    ;выполнение срочных задач
                    ;восстановление данных фоновой задачи
        SETB    EA          ;разрешение прерываний
        RETI                ;возврат к фоновой части программы
```

Остановимся подробнее на вычислении кода, заносимого в счетчик для подготовки следующего прерывания. В данном примере на вход счетчика подаются импульсы с частотой 1 МГц, а запрос на прерывание выдается в тот момент, когда его содержимое изменяется с FFFFh на 0000h. Но если вы прибавите к коду D905h код 2710h (10000 десятичное), то получите 10015h, а не 10000h. Но это не ошибка, потому что к моменту записи в счетчик младшего байта кода на вход счетчика уже поступил 21 импульс. Расчет по времени выполнения команд программы дает только 12 мкс, но еще 9 мкс тратится на переход от момента запроса прерывания к обработке прерывания. Обратите внимание еще на две особенности. Во-первых, на время обновления содержимого счетчика прерывания запрещаются. Во-вторых, сначала обновляется содержимое старшего байта. Если изменить порядок загрузки кода, то в некоторых случаях период цикла может быть отработан неправильно. Во избежание такой ошибки некоторые программисты останавливают счет на время загрузки счетчика. При правильной последовательности загрузки остановка счетчика не нужна.

В приведенных шаблонах вместо строки, относящейся к выполнению срочных задач, нужно разместить блоки счета номера цикла с распределением команд вызова различных задач в зависимости от периода и фазы их выполнения. Обычно диапазон периодов вызова задач различного назначения простирается от десятков миллисекунд до минут. В связи с разнообразием решаемых задач и большим количеством вариантов применяемых на практике способов передачи управления программам, решающим частные задачи, автору не удалось подобрать ни одного короткого и выразительного примера. В принципе существует возможность построения программы-диспетчера для распределения времени между задачами. Но универсальная программа потребляет больше ресурсов, поэтому приходится всякий раз разрабатывать синхронизирующую программу заново.

Счетчики микроконтроллера могут использоваться не только для синхронизации. Зачастую они могут использоваться для ввода и вывода информации, что позволяет исключить соответственно аналогово-цифровые и цифро-аналоговые преобразователи. Перейдем к рассмотрению этих примеров.

7.4. Программирование приема информации от датчиков

Счетчик микроконтроллера позволяет осуществить ввод информации в случае ее представления частотой импульсов. Простейшим примером является измерение угловой скорости ротора электродвигателя при помощи обтюратора, прерывающего световой поток в оптронной паре. Пусть светодиод оптронной пары включен, и накопление импульсов осуществляется в течение 0,1 с, а обтюратор содержит 24 щели. Тогда угловой скорости 1000 об/мин соответствует частота импульсов 400 Гц. Из-за отсутствия синхронизации и из-за погрешностей изготовления обтюратора в разных циклах счета при этой скорости может накопиться от 39 до 41 импульсов. Абсолютная погрешность единичного измерения, обусловленная квантованием входного сигнала, в этом случае не превышает 25 об/мин, что вполне допустимо для многих приложений. Для уменьшения этой погрешности можно увеличить время накопления, но тогда увеличивается погрешность измерения, обусловленная изменениями угловой скорости ротора (например, угловым ускорением). Если угловая скорость ротора не превышает 10000 об/мин, то в микроконтроллере она представляется числом не более 400. В этом случае для ее хранения нужно отвести два байта. Пусть измеренное значение скорости надо записать в ячейки omgl (младший байт) и omgh (старший байт).

При инициализации счетчика в старшую половину байта регистра управления режимами нужно записать Dh. Если частота прерывания равна 100 Гц, то ввод угловой скорости должен осуществляться на каждом десятом цикле. Измерение угловой скорости относится к срочным задачам, поэтому приведенный ниже пример должен располагаться в основной части программы:

```
        DEC     ten           ;счетчик для перехода от 100 Гц к 10 Гц
        MOV     A, ten        ;для проверки содержимого счетчика
        JNZ     nomes         ;переход в 9 циклах из 10
        CLR     TR1           ;остановка счетчика 1
        MOV     omgl, TL1     ;чтение мл. байта скорости
        MOV     omgh, TH1     ;чтение ст. байта скорости
        MOV     TL1, #0       ;очистка мл. байта счетчика
        MOV     TH1, #0       ;очистка ст. байта счетчика
        SETB    TR1           ;запуск счетчика 1
        MOV     ten, #10      ;запись 10 для следующего измерения
nomes:  NOP                   ;для записи метки
```

К приведенному примеру следует сделать следующие разъяснения. Команды, относящиеся к переходу от 100 Гц к 10 Гц следует использовать для обслуживания всех блоков, которые должны работать с частотой 10 Гц. В случае необходимости оптимизации программы по времени работы нужно выполнять различные блоки при разных значениях переменной ten, передавая им управление по сравнению накопителя с соответствующими константами. Команды остановки и запуска счетчика 1 в данной программе нужны потому, что из-за задержки между моментами чтения содержимого младшего и старшего байтов счетчика может появиться ошибка.

Впрочем исключение ошибки считывания содержимого счетчика при помощи его остановки может привести к погрешности измерения за счет потери входных импульсов. Ниже приведен пример чтения обоих байтов счетчика без его остановки, что необходимо для повышения точности измерений (в особенности при использовании накапливающего фильтра). Для разрешения неопределенности результатов чтения старший байт читается дважды. Между двумя чтениями содержимого старшего байта счетчика производится чтение младшего байта. Интервал между этими событиями при заданной тактовой частоте составляет всего 2 мкс, тем не менее существует вероятность несовпадения значений при первом и втором чтении старшего байта счетчика. Она тем больше, чем выше измеряемая частота. При несовпадении необходимо определить, какое из двух значений старшего байта соответствует прочитанному значению младшего байта. Изменение значения старшего байта возможно только в результате переноса из младшего. Следовательно, если при несовпадении старших байтов старший бит младшего байта равен 1, то нужно использовать результат первого чтения, а в случае 0 нужно использовать результат второго чтения.

```
        MOV   A, TH1       ;1-ое чтение ст. байта счетчика
        MOV   R2, TL1      ;чтение мл. байта счетчика
        MOV   R3, TH1      ;2-ое чтение ст. байта счетчика
        XRL   A, R3        ;проверка двух значений на совпадение
        JZ    t1ok         ;переход по совпадению значений
        MOV   A, R2
        JNB   A.7, t1ok    ;переход по переполнению мл. байта
        DEC   R3           ;уменьшение ст. байта
t1ok:   CLR   C
        MOV   A, R2        ;новое значение мл. байта счетчика
        SUBB  A, t1old     ;вычитание старого значения мл. байта
        MOV   t1old, R2    ;запоминание мл. байта счетчика
        MOV   omg1, A      ;мл. байт измерения
        MOV   A, R3        ;новое значение ст. байта счетчика
```

```
SUBB   A, thold    ;вычитание старого значения ст. байта
MOV    thold, R3   ;запоминание ст. байта счетчика
MOV    omgh, A     ;ст. байт измерения
```

После получения корректного значения содержимого счетчика измерение получается, как разность нового и старого значений, а новое значение запоминается для следующего цикла. Отметим, что вычитание дает правильный результат даже в случае однократного переполнения 16 разрядов счетчика между двумя обращениями к приведенной программе.

Кстати, стоит обратить внимание на то, что значение масштаба для представления измеренной скорости продиктовано характеристиками аппаратуры (обтюратор) и длительностью интервала накопления. Если нужно выводить измеренную скорость в об/мин на цифровую индикацию, то возможны два подхода. При первом подходе можно умножить измеренное значение на 25. Но в этом случае усложняется перевод двоичного кода в десятичный, да и две последние цифры будут недостоверными. При втором подходе можно разделить измеренное значение на 4 посредством двух сдвигов вправо. В этом случае перевод двоичного кода в десятичный получается одной операцией деления, а в младших разрядах скорости можно вывести на индикацию нули.

7.5. Программирование выдачи команд на исполнительные устройства

Программирование выдачи команд на исполнительные устройства дискретного действия типа реле или ключей не представляет проблемы. Для работы с исполнительными устройствами аналогового типа приходится использовать преобразователи цифра-аналог, но иногда можно обойтись и без них. Приведем пример использования счетчика 1 для управления приводом постоянного тока посредством широтно-импульсной модуляции (ШИМ). В этом случае при инициализации в старшую половину байта регистра управления режимами нужно записать 9h. Пусть частота циклов синхронизации как и в предыдущих примерах равна 100 Гц, а импульсы ШИМ должны выдаваться на выход P3.1. Начнем с программирования обработки запроса на прерывание. После отработки заданной длительности импульса нужно очистить заданный бит выходного порта и бит разрешения прерываний по счетчику 1.

```
.ORG   1Bh     ;адрес для прерывания по счетчику 1
CLR    P3.1    ;выключение импульса ШИМ
CLR    ET1     ;запрет прерывания по счетчику 1
RETI           ;выход из программы обработки прерывания
```

Как видите, программа обработки запроса на прерывание достаточно компактна, чтобы поместиться в пределах 8 байт. С момента выдачи запроса на прерывание до выключения импульса пройдет 12 мкс. Следовательно, программа не может выдать импульс с длительностью, меньшей некоторого порогового значения.

Теперь перейдем к программе включения импульса ШИМ. Предположим, что длительность импульса записана в ячейках pdh (старший байт) и pdl (младший байт). Ограничим длительность импульсов значениями от 13 мкс до 9996 мкс. Минимальное значение обусловлено задержками при выполнении запроса на прерывание, а максимальное — запретом на постоянное включение питания. Сначала проверим заданную длительность импульса. При длительности меньше нижней границы не будем включать импульс, а при длительности больше верхней границы включим его на максимально допустимое время.

```
       CLR   C          ;
       MOV   A, pdl     ;
       SUBB  A, #13     ;
       MOV   R0, A      ;младший байт для отсчета задержки
       MOV   A, pdh     ;
       SUBB  A, #0      ;
       MOV   R1, A      ;старший байт для отсчета задержки
       JC    zrdp       ;переход по слишком малой длительности
       SUBB  A, #27h    ;проверка на максимальную длительность
       JC    nzdp       ;переход при отсутствии насыщения
       MOV   R0, #FFh   ;младший байт для максимальной задержки
       MOV   R1, #26h   ;старший байт для максимальной задержки
nzdp:  MOV   TL1, #0    ;для задержки переноса в старший байт
       SETB  ET1        ;разрешение прерывания по счетчику 1
       CLR   C
       MOV   A, #FCh    ;младший байт максимального кода
       SUBB  A, R0      ;вычитание младшего байта длительности
       MOV   B, A       ;младший байт кода длительности
       MOV   A, #FFh    ;старший байт максимального кода
       SUBB  A, R1      ;вычитание старшего байта длительности
       MOV   TH1, A     ;старший байт кода длительности
       MOV   TL1, B     ;начало отсчета длительности импульса
       SETB  P3.1       ;включение импульса ШИМ
zrdp:  NOP              ;для записи метки
```

Задержка разрешения включения импульса относительно начала счета равна 3 мкс. Следовательно, максимальное значение кода, записываемого в счетчик 1, равно FFFCh. При этом длительность импульса ШИМ составит 13 мкс.

В некоторых из старших моделей микроконтроллеров семейства i8951 предусмотрены специальные счетчики для выдачи управляющих сигналов с широтно-импульсной модуляцией. В этом случае достаточно записать в определенные функциональные регистры коды, задающие частоту и длительность импульсов, с учетом некоторых ограничений на значения этих кодов.

7.6. Программирование ввода и вывода информации для пользователя

В большинстве изделий, управляемых микроконтроллерами, при обмене информацией с пользователем используются кнопки или клавиши для ввода и светодиодные индикаторы для вывода. В некоторых случаях для оповещения пользователя о каком-либо событии используется также звуковой сигнал. Включение или выключение звукового сигнализатора производится одной командой и посему программируется очень просто. Вывод более сложных звуковых сигналов (например, одноголосных мелодий) является любопытной, но не актуальной задачей и не рассматривается в этой книге. Перейдем к более поучительному примеру программирования ввода информации в микроконтроллер при помощи кнопок и вывода информации из микроконтроллера при помощи отдельных светодиодов и цифровых индикаторов на светодиодах.

Рассмотрим устройство с 5 группами светодиодов для вывода (4 цифровых индикатора и 5 отдельных светодиодов) и с 10 кнопками для ввода информации. Отведем по одному байту ОЗУ для каждого из каналов ввода и вывода информации:

```
        .RSECT
klv0: .DS    1              ;для ввода с кнопки 0
klv1: .DS    1              ;для ввода с кнопки 1
klv2: .DS    1              ;для ввода с кнопки 2
klv3: .DS    1              ;для ввода с кнопки 3
klv4: .DS    1              ;для ввода с кнопки 4
klv5: .DS    1              ;для ввода с кнопки 5
klv6: .DS    1              ;для ввода с кнопки 6
klv7: .DS    1              ;для ввода с кнопки 7
klv8: .DS    1              ;для ввода с кнопки 8
klv9: .DS    1              ;для ввода с кнопки 9
ind0: .DS    1              ;для вывода на индикатор 0
```

```
ind1: .DS    1              ;для вывода на индикатор 1
ind2: .DS    1              ;для вывода на индикатор 2
ind3: .DS    1              ;для вывода на индикатор 3
ind4: .DS    1              ;для вывода на индикатор 4
```

Эти ячейки предназначены для связи программ ввода и вывода с программами, решающими внутренние задачи. В принципе для ввода с кнопок можно было бы отвести по 1 бит памяти. Но в некоторых случаях бывает необходимо определить не только нажатое или отпущенное состояние кнопки, но и переход из одного состояния в другое. В этом случае для ввода с кнопок нужно отвести по 2 бит памяти. Если же в программе необходимо учитывать и длительность нажатого состояния, то приходится использовать еще по несколько битов на кнопку. Для вывода на цифровые индикаторы требуется по 1 байт памяти (7 бит на сегменты и 1 на десятичную точку).

В приведенном примере для осуществления параллельного обмена потребовалось бы 43 вывода (10 на кнопки, 28 на 4 цифровых индикатора, каждый из которых содержит по 7 сегментов, и 5 на светодиоды). В микроконтроллере имеется только 4 порта по 8 разрядов, к тому же многие из них приходится использовать для других целей. Поэтому используется последовательно-параллельный режим ввода/вывода. Это означает поочередный опрос групп кнопок и зажигание индикаторов. Притом пользователю должно казаться, что опрос всех кнопок и вывод на все индикаторы происходят одновременно. Инерция зрительного восприятия позволяет зажигать индикаторы поочередно. При выборе частоты сканирования нужно исходить из того, что каждый из индикаторов должен включаться не менее 50 раз в секунду. Следовательно для обслуживания 5 индикаторов необходимо использовать прерывания с частотой 250 Гц (период прерываний 4000 мкс). Минимальная длительность нажатия кнопки составляет около 0,1 с, поэтому частота опроса кнопок 50 Гц является достаточной.

Для прерывания с частотой 250 Гц в таймер-счетчик 0 нужно записывать код F075h. Поочередное включение питания на 5 шин возможно посредством записи единиц в 5 различных битов порта, выбранного для этой цели. Но обычно это делается более экономным способом при помощи дешифратора. Дешифратор с 3 входами позволяет коммутировать питание на 8 шинах, соответствующих кодам от 0 до 7, но мы будем использовать только коды от 0 до 4. Выберем для сканирования 3 младших бита порта 2. Вывод кода для индикации будем осуществлять через порт 0 (8 бит). Каждую из этих же шин можно использовать для опроса пары кнопок, подключив одну группу кнопок с номерами от 0 до 4 к P2.3,

а другую, с номерами от 5 до 9, к P2.4. Таким образом для ввода и вывода информации, обеспечивающей общение с пользователем, занято только 13 из 32 выводов параллельных портов. Поскольку программирование прерывания при помощи таймера-счетчика было рассмотрено ранее, перейдем к программированию синхронизации задач ввода и вывода с переключением шин сканирования индикаторов и кнопок.

При сканировании будем осуществлять изменение адреса шины в нисходящем порядке (от 4 до 0). Поскольку запись числа 4 в ячейку scan производится с частотой 50 Гц, для получения частоты 10 Гц используется еще одна ячейка — sync, содержимое которой изменяется от 5 до 1.

```
;========следующие 3 команды выполняются с частотой 250 Гц
  DEC   scan            ;уменьшение адреса шины
  MOV   A, scan         ;для проверки адреса шины
  JNB   A.7, f250       ;переход по неотрицательному значению
;========следующие 4 команды выполняются с частотой 50 Гц
  MOV   scan, #4        ;запись максимального адреса шины
  DEC   sync            ;изменение счетчика синхронизации
  MOV   A, sync         ;для проверки счетчика
  JNZ   f250            ;переход до завершения интервала 0,1 с
;========команды до метки f250 выполняются с частотой 10 Гц
  MOV   sync, #5        ;обновление содержимого счетчика
                  ;блок формирования кодов нажатия кнопок
f250:
                  ;блок вывода на индикаторы
                  ;блок проверки нажатия кнопок
```

Приведенный фрагмент обеспечивает распределение времени таким образом, что блоки вывода на индикаторы и проверки нажатия кнопок выполняются 250 раз в секунду, а блок формирования кодов нажатия кнопок выполняется 10 раз в секунду.

Рассмотрим теперь программу вывода на индикаторы, предполагая, что в ячейках ind0, ind1, ind2, ind3 и ind4 находятся необходимые для вывода коды. Для смены кода на индикаторе нужно сначала погасить все сегменты (или светодиоды), включить следующую шину и зажечь те сегменты (или светодиоды), которые соответствуют включенной шине. Притом желательно по возможности уменьшить время между гашением одного индикатора и зажиганием другого. В приведенном примере переключение вывода с одного индикатора на другой производится тремя последними командами. Предшествующие им команды обеспечивают формирование адреса шины для вывода на порт 2 и адреса для вывода содержимого ячейки на порт 0.

```
        MOV     A, scan         ;чтение адреса шины
        ADD     A, #ind0        ;адрес ячейки для вывода на индикатор
        MOV     R0, A           ;для косвенной адресации
        MOV     A, P2           ;чтение порта для переключения шины
        ANL     A, #E0h         ;выделение трех старших битов
        ORL     A, scan         ;обновление адреса шины индикаторов
        MOV     P0, #0          ;гашение светодиодов
        MOV     P2, A           ;переключение шины
        MOV     P0, @R0         ;вывод на индикатор
```

Следует обратить внимание на то, что вывод адреса шины на три младшие разряда порта 2 должен был бы производиться без изменения содержимого 5 старших разрядов этого порта. На самом деле 3 и 4 биты этого порта нужно установить в 0 для проверки нажатия кнопок.

В том же такте сканирования нужно проверить нажатие двух кнопок, присоединенных к включаемой шине. Здесь предполагается, что при нажатии кнопки сигнал с шины устанавливает соответствующий бит в 1. Если же кнопка отпущена, то импульс сканирования не проходит на вход порта 2. Запись нулей в биты 3 и 4 входного регистра порта нужна потому, что от предыдущего такта в них могли оказаться единицы. От момента переключения шины до опроса битов порта проходит время, затрачиваемое на выполнение 4 команд. Приводимая последовательность команд обеспечивает счет длительности нажатия в каждой из ячеек, отведенных для кнопок:

```
        MOV     A, scan         ;
        ADD     A, #k1v0        ;адрес ячейки для 3-го бита порта 2
        MOV     R0, A           ;
        JNB     P2.3, ch1       ;переход по не нажатой кнопке
        INC     @R0             ;счет тактов с нажатием
ch1:    ADD     A, #5           ;адрес ячейки для 4-го бита порта 2
        MOV     R0, A           ;
        JNB     P2.4, ch2       ;переход по не нажатой кнопке
        INC     @R0             ;счет тактов с нажатием
ch2:    NOP                     ;для записи метки
```

Но этим счетом обработка содержимого ячеек для ввода не заканчивается. Необходимо поверять результаты подсчета и после этого стирать содержимое счетчиков.

Счет нажатий будем вести в течение 0,1 с, что соответствует постоянной времени реакции человека. Для фильтрации ложных срабатываний кнопки будем считать кнопку нажатой, если за это время на счетчик прошло не менее 4 импульсов. Поскольку максимальное количество импульсов за 0,1 с равно 5, то для хранения этого числа достаточно 3 младших битов. Старшие биты отведем на хранение предыстории

нажатия кнопки. Для этого нужно сдвигать содержимое ячейки на 1 разряд влево и записывать нули в три младших бита. Информация из 5 старших битов может использоваться внутренней программой для определения нажатия и отпускания кнопки или даже длительности нажатия в пределах 0,5 с. Формирование кодов нажатия кнопок осуществляется следующей программой:

```
    MOV    R1, #10       ;количество кнопок
    MOV    R0, #klv0     ;адрес массива ячеек для кнопок
ag: MOV    A, @R0        ;чтение очередной ячейки для кнопок
    RL     A             ;для формирования кода предыстории
    ANL    A, #F8h       ;очистка младших битов для счета
    MOV    @R0, A        ;обновление содержимого ячейки
    INC    R0            ;адрес ячейки для следующей кнопки
    DJNZ   R1, ag        ;переход на начало цикла
```

В зависимости от способов управления изделием можно формировать коды нажатия кнопок иначе. Так, например, для приведенного метода кодирования в одном из изделий длительность нажатия кнопки была использована с целью расширения функциональных возможностей ввода числового значения при помощи кнопки. Если длительность нажатия не превышала 0,3 с, то вводимая величина изменялась на 1 при каждом нажатии. При большей длительности нажатия вводимая величина начинала изменяться на 1 каждые 0,1 с до отпускания кнопки.

В заключение отметим, что в ячейки, предназначенные для вывода на индикацию, должны записываться не числовые значения цифр, а коды, обеспечивающие зажигание нужных сегментов светодиодных индикаторов. Эти коды должны храниться в таблице, являющейся усеченным аналогом кодовых страниц компьютеров. Усечение связано с тем, что вместо матрицы хотя бы из 6×8 точек (6 байт) для каждого символа дисплея при использовании сегментных индикаторов достаточно хранить 1 байт для каждой цифры. Вообще-то индикаторы, состоящие из 7 сегментов, позволяют воспроизводить некоторое подобие букв. Но нам достаточно 10 цифр вместо 256 символов кодовой таблицы. Обычно выводы сегментов обозначаются строчными латинскими буквами. Букве “a” соответствует верхний (горизонтальный) сегмент, далее следуют в алфавитном порядке периферические сегменты по ходу часовой стрелки, а расположенный внутри горизонтальный сегмент обозначается буквой “g”. Предполагая, что эти сегменты подключены в алфавитном порядке к битам от 0 до 6, получим следующую таблицу для случая включения сегментов единицей соответствующего бита:

```
dgts: .DB   3Fh, 06h, 5Bh, 4Fh, 66h, 6Dh, 7Dh, 07h, 7Fh, 6Fh
```

Программы, решающие внутренние задачи, должны записывать в ячейки для вывода информации коды, по которым программа вывода обеспечивает зажигание определенных светодиодов. Пусть ячейкам ind0, ind1, ind2 и ind3 соответствуют цифровые индикаторы разрядов единиц, десятков, сотен и тысяч, а ячейке ind4 — отдельные светодиоды. Обновлять содержимое ячеек для отображения десятичного числа нужно сразу же после преобразования двоичного числа в десятичное.

```
MOV     DPTR, #dgts   ;адрес таблицы с кодами
                      ;десятичных цифр
MOV     A, R3         ;количество тысяч
MOVC    A, @A+DPTR    ;
MOV     ind3, A       ;код цифры для тысяч
MOV     A, R2         ;количество сотен
MOVC    A, @A+DPTR    ;
MOV     ind2, A       ;код цифры для сотен
MOV     A, R1         ;количество десятков
MOVC    A, @A+DPTR    ;
MOV     ind1, A       ;код цифры для десятков
MOV     A, R0         ;количество единиц
MOVC    A, @A+DPTR    ;
MOV     ind0, A       ;код цифры для единиц
```

Необходимо отметить, что приведенные здесь примеры относятся к частным случаям кодирования информации, зависящим от аппаратной реализации изделия. Распределение выводов по портам, используемые виды прерывания и кодирование вводимой и выводимой информации могут изменяться в зависимости от схемотехнических решений, определяемых технологией изготовления и стоимостью изделия.

Глава 8

Несколько рекомендаций о стиле программирования

8.1. Стиль программирования и использование ресурсов

Начнем с того, что современная технология программирования предусматривает определенную последовательность выполнения работ. После создания исходного текста необходимо транслировать его в объектный код и при наличии синтаксических ошибок корректировать программу до их устранения. Затем аналогичным образом повторяется компоновка объектных модулей до устранения синтаксических ошибок в связях между модулями и до приемлемого использования ресурсов памяти. Полученную в результате трансляции и компоновки исполняемую программу нужно отлаживать до устранения ошибок в семантике (смысле действий, производимых программой).

Но в рамках этой технологии программист имеет множество степеней свободы (их излишек не всегда способствуют успешному программированию). Возможности выбора алгоритма решения задачи и языка для написания исходного текста приводят к множеству вариантов. В этом плане использование Ассемблера при создании исходных текстов программ для микроконтроллеров можно считать хорошим ограничением. Система команд семейства i8051 имеет достаточно разнообразные операции и способы адресации операндов, что также предоставляет некоторую свободу действий программисту и позволяет создавать отличающиеся по стилю исполнения программы, выполняющие одну и ту же задачу.

Существование стиля обусловлено тем, что работа может быть выполнена разными способами с более или менее одинаковыми результатами. Стиль в смысле приемов и методов, используемых для повышения эффективности того или иного вида работы, очень важен. Вдобавок стиль программирования сродни также совокупности приемов, используемых для выражения мысли. Но не следует увлекаться аналогиями, ибо искусственный язык программирования предназначен для выражения не мыслей, а правил вычисления (алгоритмов). Целесообразность того или иного стилистического приема в программировании определяется экономией самых разных ресурсов на этапах разработки, изготовления и эксплуатации изделия. Таким образом стиль программирования является дополнительным фактором, оправдывающим применение микроконтроллеров для удешевления разработки, производства и эксплуатации изделий, а также для повышения удобства работы с изделием.

Критерии оптимизации программ с точки зрения экономии ресурсов должны выбираться в зависимости от конкретных обстоятельств. Задачей программиста является экономия таких ресурсов, как время работы программы и используемые объемы ОЗУ и ПЗУ. В отличие от больших проектов программирование для микроконтроллеров осуществляется как правило или одним программистом, или небольшой группой программистов. Хотя в силу этого обстоятельства опасность случившегося при вавилонском столпотворении смешения языков отпадает, хороший стиль программирования всегда полезен. Он важен для экономии времени, затрачиваемого на разработку программы, ее отладку и поддержку, необходимую в процессе усовершенствования изделия по результатам его эксплуатации или в случае введения доработок в аппаратуру.

Стиль программирования в значительной степени определяет успехи программиста и отражает его индивидуальность. Неряшливость в программировании тоже можно считать стилем, но это плохой стиль. Для улучшения стиля программисту время от времени необходимо осмысливать проделанную работы не только с точки зрения достигнутых результатов, но и с точки зрения затраченных на это средств. Усвоение хорошего стиля программирования должно идти параллельно с обучением программированию на Ассемблере. Стиль характеризует творческую сторону работы программиста, поэтому строгих правил здесь нет. Автор попытался изложить общие рекомендации, следуя которым можно быстрее добиться успеха. Однако каждый программист волен применять те или другие советы по-своему или не применять их. Важно не сотворить себе кумира (здесь автору приходится чистосердечно признаться, что в силу ряда причин его стиль программирования оставляет желать много лучшего) и по мере накопления опыта создавать удобные для себя правила и приемы программирования.

8.2. Оформление исходного текста программы

При написании исходного текста главной заботой программиста является однозначность изложения алгоритма, чтобы транслятор и компоновщик сформировали исполняемый код, соответствующий его замыслам. Можно придумать великое множество различных вариантов оформления исходного текста, дающих в результате одинаковый машинный код, то есть равноценных для микроконтроллера. Но это не значит, что все эти варианты равноценны для программиста. Оформление исходного текста не влияет на общение между программистом и машиной, но оно важно как средство выражения мыслей программиста. Стиль оформления исходного текста должен служить общению между программистами и обеспечивать читаемость программы.

Понять смысл плохо оформленного исходного текста немногим легче чем объектный или исполняемый код. Образцы плохо оформленных ассемблерных текстов можно получить из исполняемого кода при помощи специальных программ, называемых дизассемблерами. Для того, чтобы понять назначение и принцип работы таких программ, приходится проделывать очень большую работу над этими текстами. И здесь дело не только в различии образа мышления автора программы и того, кто старается разобраться в ней. Любому профессиональному программисту знакомо состояние трудности восприятия, когда через некоторое время бывает необходимо доработать какую-нибудь программу. Говоря словами великого пролетарского поэта, поначалу он даже на свою (родную!) программу "глядит, как в афишу коза". Так что хорошее оформление нужно не только для других, но и для себя. Точнее, писать программу нужно так, чтобы с этим исходным текстом было удобно работать не только другим, но и самому себе.

Основным требованием к стилю программирования является простота и ясность изложения. Весьма желательно строить программу таким образом, чтобы она занимала меньше места в памяти, расходовала меньше ячеек ОЗУ и выполнялась быстрее. Этому способствует хороший стиль программирования, хотя оптимизация требует усилий и времени. Коротко и ясно написанный исходный текст облегчает отладку и сопровождение программы. Как наставляли нас школьные учителя русского языка и литературы по поводу сочинений, при их написании нужно следовать знаменитому афоризму "чтобы словам было тесно, а мыслям — просторно". Эту заповедь с учетом специфики программирования можно распространить и на оформление исходного текста.

Весьма существенным для ясности изложения является хорошее структурирование программы. Подобно разделению текста с художественным или научным содержанием на главы, параграфы и абзацы программа также должна быть разбита на модули, подпрограммы и блоки. От выбора последовательности расположения структурных элементов зависит понимание смысла программы. В этом плане желательно располагать исходный текст таким образом, чтобы при чтении программы было легко найти данные, константы, подстановки и подпрограммы, на которые ссылаются команды и директивы, или комментарии, относящиеся к этим программным объектам. Директивы Ассемблера позволяют расположить части исходного текста, относящиеся к командам и к данным, в непосредственной близости друг к другу. Читаемость программ и уменьшение вероятности ошибок при записи текста могут быть улучшены такими средствами языка, как выражения, числовые и простые текстовые подстановки, подпрограммы и сложные текстовые подстановки.

В исходном тексте единственным средством для выражения мыслей программиста на родном языке являются комментарии. Так что они очень важны с точки зрения стиля программирования. Комментировать нужно на всех уровнях: от отдельных команд и директив до программы в целом. Комментарии ко всем структурным элементам программы используются только при необходимости и по мере возможности должны быть краткими. Комментируя на уровне команды или директивы, не нужно разъяснять то, что видно без слов по мнемокоду и используемым операндам. Необходимо объяснить смысл производимых действий, если он не ясен из комментариев к более крупным структурным частям исходного текста. Комментарии к командам и директивам должны быть самыми краткими и, как правило, помещаться в той же строке. Приведенные в книге примеры грешат избыточными построчными комментариями. Но это сделано специально с целью обучения начинающих программистов.

Комментарии на уровнях блоков лучше размещать на отдельных строках. Они могут быть более пространными и для более крупных блоков должны включать пояснения по используемым алгоритмам. В комментариях к подпрограммам и программным модулям нужно указывать их назначение, используемые ими ресурсы и сведения, необходимые для организации взаимодействия с ними других частей программы. Такого рода комментарии обычно обрамляются псевдографическими символами, что привлекает к ним внимание при чтении. Такого же рода комментарии нужны и к программе в целом. На этом уровне желательно также указывать автора программы, данные для контакта с ним и дату создания программы. Если задать эту информацию о программе как текстовую константу, то после трансляции она войдет и в исполняемый код. Тогда по содержимому ПЗУ можно найти автора программы.

Кроме того, для программы в целом может оказаться очень полезной и другая информация, записанная в исходном тексте в качестве комментариев. Очень полезно записывать в исходном тексте сведения об интерфейсе микроконтроллера с изделием, в котором он работает. Программисты, хорошо знакомые с электроникой, конечно могут работать с электрической схемой изделия. Но даже в этом случае гораздо удобнее записать в заголовке программы краткие сведения о режимах работы, характеристиках входных и выходных сигналов, обрабатываемых микроконтроллером. Включение такого рода информации в исходный текст программы полезно еще и потому, что при модернизациях изделия специалисты по электронике зачастую вводят изменения в электрическую схему, не считая нужным сообщать об этом программистам. Для программирования ввода и вывода информации необходимо также включать в программу комментарии со сведениями о взаимодействии пользователя с программой.

В процессе разработки и отладки программы нужно сохранять исходные тексты всех версий программы. Под версией программы подразумевается промежуточный вариант исходного текста, выполняющий определенный набор задач. При работе над текущей версией нежелательно вводить большое количество изменений в текст программы. Большие скачки связаны с большим риском. Каковы бы ни были изменения в исходном тексте, лучше сохранять резервную копию для отмены этих изменений. После отладки очередной версии нужно проверить исходный текст и по свежим следам дополнить программу комментариями.

8.3. Системы обозначений, выражения и простые подстановки

От общих вопросов оформления программы перейдем к частностям, которые поначалу могут показаться мелочами. Но значительная доля ошибок приходится именно на мелочи. Желательно тем или иным образом разделять обозначения зарезервированных имен от имен, назначаемых программистом. Для этого можно использовать различие между прописными и строчными буквами. В настоящей книге мнемокоды команд и директив, а также все зарезервированные слова языка записаны прописными буквами. Имеет также смысл отличать команды от директив, для этого в мнемокоде директивы лучше использовать точку. Если для записи зарезервированных имен используются прописные буквы, то имена, назначенные программистом, лучше записывать строчными буквами.

Использование мнемоники при записи имен, назначаемых программистом, затрудняется тем, что ассемблер воспринимает только латинский алфавит. В связи с этим только некоторая часть программистов может щеголять знанием английского языка. У англоязычных программистов принято обозначать операнды существительными, а подпрограммы — глаголами. Несведущие в английском языке с успехом пользуются русскими словами, записанными на латинице. Программист может изобретать и использовать свои мнемонические правила, укладывающиеся в синтаксические ограничения языка. Эти правила позволяют уменьшить вероятность ошибок при записи имен и следить за смыслом выполняемых действий. Так, например, имена разных групп операндов могут иметь различную длину. При использовании переменной, состоящей из двух байтов, полезно использовать для них имена с одинаковым корнем и двумя разными суффиксами: lo (low — нижний) для младшего байта и hi (high — верхний) для старшего. Для того чтобы отличать битовые переменные, можно использовать префикс f_ (флаг). Такого рода правила не противоречат синтаксису Ассемблера и существенно облегчают работу программиста. Вводя свои мнемонические правила, следует записывать их в комментариях. Необходимо следить за непротиворечивостью этих правил, дабы не запутаться самому. При выборе своего стиля программист должен предусматривать меры по уменьшению вероятности разного рода ошибок при вводе исходного текста и его коррекции. Некоторые широко используемые приемы такого рода рассмотрены ниже.

Необходимы определенные правила и при выборе длины имени. Некоторым программистам нравится использовать короткие имена, но чрезмерное увлечение сокращением объема исходного текста в байтах не всегда оправдано. Умеренная избыточность в длинах имен полезна, так как она уменьшает вероятность совпадения с другим именем в случае опечатки. Транслятор способен выявлять опечатки только в том случае, когда результат ошибки не приводит к совпадению с другими именами в программе. В том случае, когда "время жизни" переменной ограничивается несколькими командами или небольшим блоком команд, целесообразно использовать для нее регистр общего назначения или ячейку памяти с однобуквенным именем. Полезно также использовать локальные метки для передачи управления в пределах небольшого блока. В остальных случаях общепринятые правила выбора имен рекомендуют назначать короткие имена объектам в модуле и длинные — объектам, используемым для связи между модулями.

Еще одной группой мелочей, которые могут добавить массу неприятностей, являются константы. Во-первых, при записи констант программист

может допустить ошибки в вычислениях. Во избежание этого нужно пользоваться теми возможностями, которые предоставляет ассемблер для перехода от одной системы счисления к другой и для вычисления выражений. Поэтому нужно записывать константы в таком виде, чтобы смысл их был понятен программисту, а пересчет их значений во внутренний код будет произведен автоматически. Числовые константы удобнее записывать в десятичном виде. Логические константы удобнее записывать в двоичном или шестнадцатеричном виде. Символьные константы удобнее записывать в виде строки. Если константа получается как результат каких-то арифметических вычислений, то лучше записать ее в виде выражения, объяснив в комментарии смысл его операндов. Такой стиль записи констант облегчает поиск ошибок, чтение программы и ее модернизацию. Во-вторых, при записи одной и той же константы в разные операторы возможна ошибка (описка). Во избежание этого следует использовать подстановки.

Числовые и простые текстовые подстановки являются весьма полезными с точки зрения борьбы с опечатками. Их эффективность проявляется при создании исходных текстов, но еще более они нужны при коррекции программ. Если в процессе создания программы еще можно (хотя и трудно) уследить за числовыми или логическими константами, записанными в разных командах, то сделать это при коррекции практически невозможно. Так что настоятельно рекомендуется использовать поименованные константы, определяемые числовыми подстановками. Их использование полностью исключает возможность ошибочной записи разных значений одной и той же константы в разные операторы.

Пользу простых текстовых подстановок можно продемонстрировать на следующем примере. При записи исходного текста программы, использующей регистры общего назначения, очень легко перепутать номера регистров. Ведь транслятор не может обнаружить ошибочный ввод цифры в имени регистра, если эта цифра находится в пределах от 0 до 7. А если программист при помощи простой текстовой подстановки задаст свои мнемонические обозначения регистрам микроконтроллера, это существенно уменьшит вероятность необнаруженных опечаток.

8.4. Применение подпрограмм и сложных текстовых подстановок

Одним из способов снижения трудоемкости программирования является заимствование структурных единиц из отработанной программы для

вновь создаваемых. Такого рода стилистический прием можно считать не плагиатом, а цитированием. Заимствование самой крупной структурной единицы (объектного модуля) производится на этапе компоновки. На этапе создания исходного текста можно, уподобившись нерадивому школьнику, списать из одной программы в другую некоторую последовательность команд, но это плохой стиль. Хорошим стилем является заимствование на уровне подпрограмм и сложных текстовых подстановок. Для заимствования на уровне подпрограмм удобно использовать библиотеки, а для заимствования на уровне сложных текстовых подстановок — макроопределения во включаемых в исходный текст файлах.

При разработке программ на Ассемблере большое количество синтаксических ошибок связано с метками. Поэтому при разработке языков программирования высокого уровня были предприняты усилия для изгнания меток как средства управления потоком команд. Программирование без меток было названо структурным и оказалось очень удобным. Хотя Ассемблер не позволяет обойтись без меток, использование таких структурных единиц как модули, подпрограммы и сложные текстовые уменьшают вероятность синтаксических ошибок. Использование в программе разного рода структурных единиц также является стилистическим приемом, применение которого определяется как объективными обстоятельствами, так и “вкусом” программиста.

Удобство применения подпрограммы как способа структурирования программы настолько привлекательно, что некоторые программисты оформляют набор команд как подпрограмму даже в случае его однократного использования программой. Применение подпрограмм становится тем более оправданным, когда они используются программой многократно. Стандартизация подпрограмм, например, для вычислений при представлении чисел несколькими байтами или для использовании драйверов разнообразных устройств ввода/вывода позволяет существенно сэкономить время разработки и отладки программ. Применение библиотечных подпрограмм целесообразно с точки зрения экономии времени разработки и отладки программы. В этом случае подпрограммы хранятся в библиотеке и включаются в исполняемую программу на этапе компоновки. Большинство приведенных в книге примеров может быть оформлено как подпрограммы. Для этого необходимо в начале соответствующего блока записать метку, а в конце — команду возврата из подпрограммы. Но этого недостаточно, так как при разработке подпрограмм нужно соблюдать простые правила, обеспечивающие отсутствие помех работе вызывающей программы.

Выполняя полезную работу, подпрограмма не должна вредить, то есть портить содержимое ячеек ОЗУ, используемых вызывающей программой. Программисту нужно использовать одни и те же правила обмена информацией между подпрограммой и вызывающей программой. Эти правила должны учитывать как особенности архитектуры микроконтроллера, так и функциональное назначение подпрограмм. В связи с ограничениями на адресное пространство стека в качестве рабочих ячеек подпрограммы удобнее всего использовать регистры общего назначения. Если подпрограмма должна работать с другими ячейками ОЗУ, то целесообразнее всего использовать косвенную адресацию, записывая адреса в регистры R0 и R1. В этом случае подпрограмма может работать только с двумя группами ячеек ОЗУ. При использовании остальных функциональных регистров их содержимое должно сохраняться в стеке на время выполнения подпрограммы и восстанавливаться перед выходом из подпрограммы. Исключения из этого правила должны оговариваться в описании подпрограммы. Для удобства использования подпрограмм в заголовке каждой из них следует записать комментарии с описанием назначения, форматами и адресами входных и выходных данных и с информацией об использовании функциональных регистров.

Рассмотрев очевидные достоинства применения подпрограмм, перейдем к их недостаткам. Обращение к подпрограмме приводит к затратам адресного пространства стека для запоминания адреса возврата и сохранения содержимого функциональных регистров. В связи с ограниченным объемом ОЗУ это затрудняет широкое использование подпрограмм в микроконтроллерах. Для перехода к подпрограмме и возврата из нее, а также для записи операндов в стандартные ячейки подпрограммы и чтения результатов работы из них расходуется время, поэтому использование подпрограмм для реализации слишком простых вычислений нецелесообразно. Применение подпрограмм удобно тогда, когда ограничения на объем ПЗУ более существенны, чем ограничения на быстродействие программы и/или на объем ОЗУ. Они также более удобны тем, что их можно наладить и оттранслировать заранее.

В отличие от подпрограмм сложные текстовые подстановки (макросы) работают без передачи управления и могут использовать не только косвенную, но и непосредственную адресацию к любым ячейкам ОЗУ. Их применение удобно тогда, когда ограничения на объем ПЗУ менее существенны, чем ограничения на быстродействие программы и/или на объем ОЗУ. При прочих равных условиях замена подпрограмм на сложные текстовые подстановки приводит к увеличению объема исполняемой программы. Выбор между использованием подпрограмм и сложных

текстовых подстановок чаще всего (при отсутствии строгих ограничений по ресурсам) является делом вкуса программиста. Но в некоторых случаях выбор бывает предопределен ограничением ресурсов или спецификой программируемого алгоритма.

Применение сложных текстовых подстановок требует хорошего знания синтаксических правил и выражений для подстановок, чтобы при подстановке фактических параметров получились такие команды, которые нужны программисту. Использование конструкций такого рода в языках высокого уровня часто приводит к побочным эффектам, обусловленным непониманием формализма синтаксических правил. Соответствующие примеры приводятся в учебниках по языку Си, и именно по этой причине не рекомендуется использовать макросы. Впрочем, в Ассемблере ситуация облегчается возможностью полной проверки результата подстановки по листингу.

Заключение

Оглядываясь на проделанную работу, автор удовлетворен тем, что не пошел по соблазнительному пути представления читателям готовых блоков из программ, успешно работающих в составе различных изделий. Сборник рецептов не принес бы пользу читателям, так как невозможно заготовить решения всех задач, которые могут встретиться программисту. Такое издание могло бы только потешить авторское самолюбие своим объемом. Гораздо полезнее учебное пособие, которое не должно давать готовых рецептов. Оно должно показывать разнообразие способов решения типовых задач и учить поиску целесообразных решений. Возможно, кому-нибудь из читателей не понравится несколько расплывчатый стиль изложения, обусловленный некоторой склонностью автора к философствованию и резонерству. Автор по мере возможности сдерживал такие порывы, "наступая на горло собственной песне". Но по завершении работы он считает себя свободным от оков жанра, решив в меру своей испорченности предаться рассуждениям о ее целесообразности.

Каждая книга так или иначе должна служить поиску истины в какой-либо из сфер жизни и деятельности человека. В этом плане хотелось бы высказаться о паре латинских изречений "Истина в вине" и "В споре рождается истина". Что касается первого изречения, то оно относится к тем, кто ищет ИСТИНУ в конечной инстанции. Автор не верит в существование таковой, а также является дилетантом по части средства поисков, отдавая явное предпочтение книгам, в которых вместо всеобъемлющей истины содержатся только ее частные составляющие.

При обсуждении второго изречения нужно уточнить понятие о споре, как о словесном состязании культурных людей. Некоторые из господ ученых очень любят дискуссии. Плодом спора чаще всего являются ссоры, взаимная неприязнь, интриги, а то и непристойное поведение, заканчивающееся неприличными выкриками и рукоприкладством.

Странно, что древние римляне ничего не говорили о зачатии истины (то ли ничего не знали об этом, то ли утаили свои знания). Хотя истина является нематериальной субстанцией, все же до рождения ее надо зачать

и выносить. Зачинается она по-видимому в голове из идей, которые (как говорят) носятся в воздухе. Но идеи могут носиться всякие, да и головы тоже разные бывают. Так что до истины может развиться не всякая идея и не в любой голове. Что же касается рождения истины, то автор убежден, что это дело настолько интимное, что оно должно совершаться только вдвоем, в процессе дружественной беседы. Притом беседующие должны быть в хорошем телесном и душевном состоянии и в благосклонном настроении друг к другу. Поэтому нужно заменить классическую формулировку на новую: "Истина рождается в беседе". К сожалению автор не учился в классической гимназии, иначе перевел бы это изречение на латинский язык в назидание древним римлянам. Но автор адресует свою редакцию древнего изречения прежде всего читателям. Для успешной работы программисту (и не только ему!) нужно беседовать, а не спорить с другими участниками совместной работы.

Теперь читателям должно быть понятно стремление автора построить изложение в форме заочной беседы (если таковая в принципе возможна). Интересно, что в стихотворении "Герой", имеющем эпиграф "Что есть истина?", А.С.Пушкин описывает беседу Друга с Поэтом, заявившим:

Тьмы низких истин мне дороже
Нас возвышающий обман...

Так что автор испытал чувство умиления от того, что своим умом дошел до мысли о продуктивности дружественных бесед через два века после гения. Слово "тьма" в приведенной цитате является именем числительным (десять тысяч), а смысл имени прилагательного "низких" следует понимать как частных, не всеобъемлющих. Что касается возвышающего обмана, то с точки зрения темы стихотворения предпочтение Поэта достойно уважения. Тема настоящей книги не столь значительна, поэтому соотношение между десятками тысяч низких истин и каким бы то ни было обманом обратное. Поскольку главным героем научно-технических трудов является некоторый набор частных истин, то по законам жанра значимость прочих персонажей как бы принижается. К прочим относятся читатели, упоминаемые в третьем лице множественного числа, и автор, упоминаемый в третьем лице единственного числа. На самом деле изложенные в этих трудах частные истины ничего ие значат без трудов читателей и автора, а посему в конце книги эту условность, как бы отстраняющую автора от читателей, можно отбросить.

Не опасаясь прослыть последней буквой в азбуке, Я желаю ВАМ успехов как в программировании, так и во всех ВАШИХ добрых делах!

ПРИЛОЖЕНИЯ

Приложение 1

Перечень команд семейства i8051, упорядоченный по кодам операций

00	NOP	
01 FF	AJMP	frst
02 00 FF	LJMP	frst
03	RR	A
04	INC	A
05 F0	INC	B
06	INC	@R0
07	INC	@R1
08	INC	R0
09	INC	R1
0A	INC	R2
0B	INC	R3
0C	INC	R4
0D	INC	R5
0E	INC	R6
0F	INC	R7
10 D5 3F	JBC	F0, lbl1
11 FF	ACALL	frst
12 00 FF	LCALL	frst
13	RRC	A
14	DEC	A
15 F0	DEC	B
16	DEC	@R0

17		DEC	@R1
18		DEC	R0
19		DEC	R1
1A		DEC	R2
1B		DEC	R3
1C		DEC	R4
1D		DEC	R5
1E		DEC	R6
1F		DEC	R7
20	D6 29	JB	AC, lbl1
21	FE	AJMP	scnd
22		RET	
23		RL	A
24	FF	ADD	A, #255
25	E0	ADD	A, A
26		ADD	A, @R0
27		ADD	A, @R1
28		ADD	A, R0
29		ADD	A, R1
2A		ADD	A, R2
2B		ADD	A, R3
2C		ADD	A, R4
2D		ADD	A, R5
2E		ADD	A, R6
2F		ADD	A, R7
30	D5 14	JNB	F0, lbl1
31	FE	ACALL	scnd
32		RETI	
33		RLC	A
34	FF	ADDC	A, #FFh
35	D0	ADDC	A, PSW
36		ADDC	A, @R0
37		ADDC	A, @R1
38		ADDC	A, R0
39		ADDC	A, R1
3A		ADDC	A, R2
3B		ADDC	A, R3
3C		ADDC	A, R4
3D		ADDC	A, R5
3E		ADDC	A, R6
3F		ADDC	A, R7

```
40 00            JC      lbl1
41 FD      lbl1: AJMP    thrd
42 80            ORL     P0, A
43 90 10         ORL     P1, #10h
44 0F            ORL     A, #0Fh
45 A0            ORL     A, P2
46               ORL     A, @R0
47               ORL     A, @R1
48               ORL     A, R0
49               ORL     A, R1
4A               ORL     A, R2
4B               ORL     A, R3
4C               ORL     A, R4
4D               ORL     A, R5
4E               ORL     A, R6
4F               ORL     A, R7
50 00            JNC     lbl2
51 FD      lbl2: ACALL   thrd
52 B0            ANL     P3, A
53 B8 07         ANL     IP, #07h
54 F0            ANL     A, #F0h
55 80            ANL     A, P0
56               ANL     A, @R0
57               ANL     A, @R1
58               ANL     A, R0
59               ANL     A, R1
5A               ANL     A, R2
5B               ANL     A, R3
5C               ANL     A, R4
5D               ANL     A, R5
5E               ANL     A, R6
5F               ANL     A, R7
60 00            JZ      lbl3
61 FC      lbl3: AJMP    frth
62 80            XRL     P0, A
63 8A FF         XRL     TL0, #FFh
64 0F            XRL     A, #0Fh
65 8C            XRL     A, TH0
66               XRL     A, @R0
67               XRL     A, @R1
68               XRL     A, R0
```

```
69                  XRL     A, R1
6A                  XRL     A, R2
6B                  XRL     A, R3
6C                  XRL     A, R4
6D                  XRL     A, R5
6E                  XRL     A, R6
6F                  XRL     A, R7
70 00               JNZ     lbl4
71 FC        lbl4:  ACALL   frth
72 D2               ORL     C, OV
73                  JMP     @A+DPTR
74 63               MOV     A, #99
75 F0 64            MOV     B, #100
76 00               MOV     @R0, #00h
77 01               MOV     @R1, #01h
78 02               MOV     R0, #02h
79 03               MOV     R1, #03h
7A 04               MOV     R2, #04h
7B 05               MOV     R3, #05h
7C 06               MOV     R4, #06h
7D 07               MOV     R5, #07h
7E 08               MOV     R6, #08h
7F 09               MOV     R7, #09h
80 00               SJMP    lbl5
81 FB        lbl5:  AJMP    ffth
82 D0               ANL     C, P
83                  MOVC    A, @A+PC
84                  DIV     AB
85 F0 99            MOV     SBUF, B
86 B8               MOV     IP, @R0
87 A8               MOV     IE, @R1
88 98               MOV     SCON, R0
89 89               MOV     TMOD, R1
8A 88               MOV     TCON, R2
8B 81               MOV     SP, R3
8C B0               MOV     P3, R4
8D A0               MOV     P2, R5
8E 90               MOV     P1, R6
8F 80               MOV     P0, R7
90 12 34            MOV     DPTR, #1234h
91 FB               ACALL   ffth
```

```
92 D5                    MOV     F0, C
93                       MOVC    A, @A+DPTR
94 0F                    SUBB    A, #15
95 F0                    SUBB    A, B
96                       SUBB    A, @R0
97                       SUBB    A, @R1
98                       SUBB    A, R0
99                       SUBB    A, R1
9A                       SUBB    A, R2
9B                       SUBB    A, R3
9C                       SUBB    A, R4
9D                       SUBB    A, R5
9E                       SUBB    A, R6
9F                       SUBB    A, R7
A0 F0                    ORL     C, /B.0
A1 FA                    AJMP    sxth
A2 F7                    MOV     C, B.7
A3                       INC     DPTR
A4                       MUL     AB
             ;команды с кодом операции A5 нет
A6 8A                    MOV     @R0, TL0
A7 8C                    MOV     @R1, TH0
A8 8B                    MOV     R0, TL1
A9 8D                    MOV     R1, TH1
AA 82                    MOV     R2, DPL
AB 83                    MOV     R3, DPH
AC 80                    MOV     R4, P0
AD 90                    MOV     R5, P1
AE A0                    MOV     R6, P2
AF B0                    MOV     R7, P3
B0 D2                    ANL     C, /OV
B1 FA                    ACALL   sxth
B2 D5                    CPL     F0
B3                       CPL     C
B4 1F 23                 CJNE    A, #31, lbl6
B5 F0 20                 CJNE    A, B, lbl6
B6 01 1D                 CJNE    @R0, #01, lbl6
B7 02 1A                 CJNE    @R1, #02, lbl6
B8 03 17                 CJNE    R0, #03, lbl6
B9 0E 14                 CJNE    R1, #14, lbl6
BA 19 11                 CJNE    R2, #25, lbl6
```

```
BB 24 0E            CJNE   R3, #36, lbl6
BC 2F 0B            CJNE   R4, #47, lbl6
BD 3A 08            CJNE   R5, #58, lbl6
BE 45 05            CJNE   R6, #69, lbl6
BF 46 02            CJNE   R7, #70, lbl6
C0 E0               PUSH   A
C1 F9       lbl6:   AJMP   svth
C2 AF               CLR    EA
C3                  CLR    C
C4                  SWAP   A
C5 F0               XCH    A, B
C6                  XCH    A, @R0
C7                  XCH    A, @R1
C8                  XCH    A, R0
C9                  XCH    A, R1
CA                  XCH    A, R2
CB                  XCH    A, R3
CC                  XCH    A, R4
CD                  XCH    A, R5
CE                  XCH    A, R6
CF                  XCH    A, R7
D0 F0               POP    B
D1 F9       lbl7:   ACALL  svth
D2 AF               SETB   EA
D3                  SETB   C
D4                  DA     A
D5 82 F7            DJNZ   DPL, lbl7
D6                  XCHD   A, @R0
D7                  XCHD   A, @R1
D8 F3               DJNZ   R0, lbl7
D9 F1               DJNZ   R1, lbl7
DA EF               DJNZ   R2, lbl7
DB ED               DJNZ   R3, lbl7
DC EB               DJNZ   R4, lbl7
DD E9               DJNZ   R5, lbl7
DE E7               DJNZ   R6, lbl7
DF E5               DJNZ   R7, lbl7
E0                  MOVX   A, @DPTR
E1 F8               AJMP   egth
E2                  MOVX   A, @R0
E3                  MOVX   A, @R1
```

```
E4                              CLR     A
E5 99                           MOV     A, SBUF
E6                              MOV     A, @R0
E7                              MOV     A, @R1
E8                              MOV     A, R0
E9                              MOV     A, R1
EA                              MOV     A, R2
EB                              MOV     A, R3
EC                              MOV     A, R4
ED                              MOV     A, R5
EE                              MOV     A, R6
EF                              MOV     A, R7
F0                              MOVX    @DPTR, A
F1 F8                           ACALL   egth
F2                              MOVX    @R0, A
F3                              MOVX    @R1, A
F4                              CPL     A
F5 99                           MOV     SBUF, A
F6                              MOV     @R0, A
F7                              MOV     @R1, A
F8                              MOV     R0, A
F9                              MOV     R1, A
FA                              MOV     R2, A
FB                              MOV     R3, A
FC                              MOV     R4, A
FD                              MOV     R5, A
FE                              MOV     R6, A
FF                              MOV     R7, A
```

Приложение 2

Сводка команд i8051 в алфавитном порядке

```
ACALL    n                 ;ПЕРЕХОД К ПОДПРОГРАММЕ ближний
ADD[C]   A, {#|R|@|d}      ;СЛОЖЕНИЕ [С УЧЕТОМ ПЕРЕНОСА]
AJMP     n                 ;ПЕРЕХОД БЕЗУСЛОВНЫЙ ближний
ANL      A, {#|R|@|d}      ;И
```

```
ANL         d, {A|#}         ;И
ANL         C, [/]b          ;И с [обратным] кодом бита
CJNE        A, {#|d}, s      ;ПЕРЕХОД ПО НЕРАВЕНСТВУ короткий
CJNE        {R|@}, #, s      ;ПЕРЕХОД ПО НЕРАВЕНСТВУ короткий
CLR         {A|b|C}          ;ОЧИСТКА {накопителя|бита}
CPL         {A|b|C}          ;ПОЛУЧЕНИЕ ОБРАТНОГО КОДА
                             ;{накопителя|бита}
DA          A                ;КОРРЕКЦИЯ двоично-десятичного
                             ;кода
DEC         {A|R|@|d}        ;УМЕНЬШЕНИЕ НА 1
DIV         AB               ;ДЕЛЕНИЕ
DJNZ        {R|d}, s         ;УМЕНЬШЕНИЕ НА 1 И ПЕРЕХОД ПО НЕ
                             ;0 короткий
INC         {A|R|@|d}        ;УВЕЛИЧЕНИЕ НА 1
INC         D                ;УВЕЛИЧЕНИЕ НА 1
J{NB|B[C]} b, s              ;ПЕРЕХОД ПО {0|1[с очисткой]}
                             ;короткий
J[N]C       s                ;ПЕРЕХОД ПО 1 [0] ПЕРЕНОСА короткий
JMP         @A+D             ;БЕЗУСЛОВНЫЙ ПЕРЕХОД КОСВЕННЫЙ
J[N]Z       s                ;ПЕРЕХОД ПО НУЛЮ [НЕ НУЛЮ] короткий
LCALL       f                ;ПЕРЕХОД К ПОДПРОГРАММЕ дальний
LJMP        f                ;БЕЗУСЛОВНЫЙ ПЕРЕХОД дальний
MOV         A, {#|R|@|d}     ;ПЕРЕСЫЛКА
MOV         d, {A|#|R|@|d}   ;ПЕРЕСЫЛКА
MOV         {R|@}, {A|#|d}   ;ПЕРЕСЫЛКА
MOV         D, #             ;ПЕРЕСЫЛКА
MOV         {b,C|C,b}        ;ПЕРЕСЫЛКА бита
MOVC        A, @A+{D|P}      ;ПЕРЕСЫЛКА из ПЗУ
MOVX        A, {@D|@}        ;ПЕРЕСЫЛКА из внешнего ОЗУ
MOVX        {@D|@}, A        ;ПЕРЕСЫЛКА во внешнее ОЗУ
MUL         AB               ;УМНОЖЕНИЕ
NOP                          ;НЕТ ОПЕРАЦИИ
ORL         A, {#|R|@|d}     ;ИЛИ
ORL         d, {A|#}         ;ИЛИ
ORL         C, [/]b          ;ИЛИ с [обратным] кодом бита
POP         d                ;ЧТЕНИЕ ИЗ СТЕКА
PUSH        d                ;ЗАПИСЬ В СТЕК
RET[I]                       ;ВОЗВРАТ [ИЗ ПРЕРЫВАНИЯ]
R{L|R}[C]   A                ;СДВИГ {ВЛЕВО|ВПРАВО}[через бит
                             ;переноса]
SETB        {b|C}            ;ЗАПИСЬ 1 в бит
SJMP        s                ;БЕЗУСЛОВНЫЙ ПЕРЕХОД короткий
SUBB        A, {#|R|@|d}     ;ВЫЧИТАНИЕ С УЧЕТОМ ЗАЙМА
SWAP        A                ;ОБМЕН ПОЛУБАЙТОВ
XCH         A, {R|@|d}       ;ОБМЕН БАЙТОВ
```

```
XCHD        A, @            ;ОБМЕН МЛАДШИХ ПОЛУБАЙТОВ МЕЖДУ
БАЙТАМИ
XRL         A, {#|R|@|d}    ;ИСКЛЮЧАЮЩЕЕ ИЛИ
XRL         d, {A|#}        ;ИСКЛЮЧАЮЩЕЕ ИЛИ
```

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ

{ | }	обязателен выбор одного из вариантов в скобках
[]	использование варианта при необходимости
#	операнд записан в ПЗУ (в соответствии с размером приемника)
@	косвенная адресация через регистр R0 или R1
@A+D	косвенная адресация через сумму DPTR и накопителя
@A+P	косвенная адресация через сумму счетчика команд и накопителя
@D	косвенная адресация через DPTR
A	операндом является содержимое накопителя
AB	операнды содержатся в накопителе и регистре B
C	операндом является содержимое бита переноса
D	операндом является содержимое регистра DPTR
R	операндом является содержимое регистра (от R0 до R7)
b	операндом является адресуемый бит
d	операндом является адресуемый байт
f	адрес в ПЗУ от 0 до 64 Кбайт
n	адрес в текущей странице ПЗУ (размер страницы 4 Кбайт)
s	смещение относительно адреса команды от -128 до +127

Оглавление